Is Parallel Programming Hard, And, If So, What Can You Do About It?

Edited by:

Paul E. McKenney
Facebook
paulmck@kernel.org

March 14, 2021
Second Edition, Release Candidate 11

www.dbooks.org

Legal Statement

This work represents the views of the editor and the authors and does not necessarily represent the view of their respective employers.

Trademarks:

- IBM, z Systems, and PowerPC are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.
- Linux is a registered trademark of Linus Torvalds.
- Intel, Itanium, Intel Core, and Intel Xeon are trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.
- Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.
- MIPS is a registered trademark of Wave, Inc. in the United States and other countries.
- SPARC is a registered trademark of SPARC International, Inc. Products bearing SPARC trademarks are based on an architecture developed by Sun Microsystems, Inc.
- Other company, product, and service names may be trademarks or service marks of such companies.

The non-source-code text and images in this document are provided under the terms of the Creative Commons Attribution-Share Alike 3.0 United States license.[1] In brief, you may use the contents of this document for any purpose, personal, commercial, or otherwise, so long as attribution to the authors is maintained. Likewise, the document may be modified, and derivative works and translations made available, so long as such modifications and derivations are offered to the public on equal terms as the non-source-code text and images in the original document.

Source code is covered by various versions of the GPL.[2] Some of this code is GPLv2-only, as it derives from the Linux kernel, while other code is GPLv2-or-later. See the comment headers of the individual source files within the CodeSamples directory in the git archive[3] for the exact licenses. If you are unsure of the license for a given code fragment, you should assume GPLv2-only.

Combined work © 2005–2021 by Paul E. McKenney. Each individual contribution is copyright by its contributor at the time of contribution, as recorded in the git archive.

[1] `https://creativecommons.org/licenses/by-sa/3.0/us/`
[2] `https://www.gnu.org/licenses/gpl-2.0.html`
[3] `git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git`

Contents

1 How To Use This Book **1**
- 1.1 Roadmap . . . 1
- 1.2 Quick Quizzes . . . 2
- 1.3 Alternatives to This Book . . . 2
- 1.4 Sample Source Code . . . 4
- 1.5 Whose Book Is This? . . . 4

2 Introduction **7**
- 2.1 Historic Parallel Programming Difficulties . . . 7
- 2.2 Parallel Programming Goals . . . 8
  - 2.2.1 Performance . . . 9
  - 2.2.2 Productivity . . . 10
  - 2.2.3 Generality . . . 10
- 2.3 Alternatives to Parallel Programming . . . 12
  - 2.3.1 Multiple Instances of a Sequential Application . . . 12
  - 2.3.2 Use Existing Parallel Software . . . 12
  - 2.3.3 Performance Optimization . . . 13
- 2.4 What Makes Parallel Programming Hard? . . . 13
  - 2.4.1 Work Partitioning . . . 14
  - 2.4.2 Parallel Access Control . . . 14
  - 2.4.3 Resource Partitioning and Replication . . . 15
  - 2.4.4 Interacting With Hardware . . . 15
  - 2.4.5 Composite Capabilities . . . 15
  - 2.4.6 How Do Languages and Environments Assist With These Tasks? 16
- 2.5 Discussion . . . 16

3 Hardware and its Habits **17**
- 3.1 Overview . . . 17
  - 3.1.1 Pipelined CPUs . . . 17
  - 3.1.2 Memory References . . . 19
  - 3.1.3 Atomic Operations . . . 19
  - 3.1.4 Memory Barriers . . . 20
  - 3.1.5 Cache Misses . . . 20
  - 3.1.6 I/O Operations . . . 20
- 3.2 Overheads . . . 21
  - 3.2.1 Hardware System Architecture . . . 21
  - 3.2.2 Costs of Operations . . . 22
  - 3.2.3 Hardware Optimizations . . . 24

Edition.2-rc11

www.dbooks.org

3.3 Hardware Free Lunch? . . . 25
3.3.1 3D Integration . . . 26
3.3.2 Novel Materials and Processes . . . 26
3.3.3 Light, Not Electrons . . . 26
3.3.4 Special-Purpose Accelerators . . . 27
3.3.5 Existing Parallel Software . . . 27
3.4 Software Design Implications . . . 27

4 Tools of the Trade 29
4.1 Scripting Languages . . . 29
4.2 POSIX Multiprocessing . . . 30
4.2.1 POSIX Process Creation and Destruction . . . 30
4.2.2 POSIX Thread Creation and Destruction . . . 31
4.2.3 POSIX Locking . . . 32
4.2.4 POSIX Reader-Writer Locking . . . 34
4.2.5 Atomic Operations (GCC Classic) . . . 36
4.2.6 Atomic Operations (C11) . . . 36
4.2.7 Atomic Operations (Modern GCC) . . . 37
4.2.8 Per-Thread Variables . . . 37
4.3 Alternatives to POSIX Operations . . . 37
4.3.1 Organization and Initialization . . . 37
4.3.2 Thread Creation, Destruction, and Control . . . 38
4.3.3 Locking . . . 39
4.3.4 Accessing Shared Variables . . . 40
4.3.5 Atomic Operations . . . 46
4.3.6 Per-CPU Variables . . . 46
4.4 The Right Tool for the Job: How to Choose? . . . 47

5 Counting 49
5.1 Why Isn't Concurrent Counting Trivial? . . . 49
5.2 Statistical Counters . . . 51
5.2.1 Design . . . 51
5.2.2 Array-Based Implementation . . . 51
5.2.3 Per-Thread-Variable-Based Implementation . . . 52
5.2.4 Eventually Consistent Implementation . . . 54
5.2.5 Discussion . . . 55
5.3 Approximate Limit Counters . . . 55
5.3.1 Design . . . 55
5.3.2 Simple Limit Counter Implementation . . . 56
5.3.3 Simple Limit Counter Discussion . . . 59
5.3.4 Approximate Limit Counter Implementation . . . 60
5.3.5 Approximate Limit Counter Discussion . . . 61
5.4 Exact Limit Counters . . . 61
5.4.1 Atomic Limit Counter Implementation . . . 61
5.4.2 Atomic Limit Counter Discussion . . . 64
5.4.3 Signal-Theft Limit Counter Design . . . 64
5.4.4 Signal-Theft Limit Counter Implementation . . . 65
5.4.5 Signal-Theft Limit Counter Discussion . . . 67
5.4.6 Applying Exact Limit Counters . . . 68
5.5 Parallel Counting Discussion . . . 68

5.5.1 Parallel Counting Performance . . . 69
5.5.2 Parallel Counting Specializations . . . 69
5.5.3 Parallel Counting Lessons . . . 70

6 Partitioning and Synchronization Design 73
6.1 Partitioning Exercises . . . 73
6.1.1 Dining Philosophers Problem . . . 73
6.1.2 Double-Ended Queue . . . 75
6.1.3 Partitioning Example Discussion . . . 80
6.2 Design Criteria . . . 80
6.3 Synchronization Granularity . . . 82
6.3.1 Sequential Program . . . 82
6.3.2 Code Locking . . . 82
6.3.3 Data Locking . . . 84
6.3.4 Data Ownership . . . 85
6.3.5 Locking Granularity and Performance . . . 85
6.4 Parallel Fastpath . . . 87
6.4.1 Reader/Writer Locking . . . 88
6.4.2 Hierarchical Locking . . . 88
6.4.3 Resource Allocator Caches . . . 88
6.5 Beyond Partitioning . . . 92
6.5.1 Work-Queue Parallel Maze Solver . . . 92
6.5.2 Alternative Parallel Maze Solver . . . 93
6.5.3 Performance Comparison I . . . 95
6.5.4 Alternative Sequential Maze Solver . . . 96
6.5.5 Performance Comparison II . . . 96
6.5.6 Future Directions and Conclusions . . . 97
6.6 Partitioning, Parallelism, and Optimization . . . 98

7 Locking 99
7.1 Staying Alive . . . 99
7.1.1 Deadlock . . . 99
7.1.2 Livelock and Starvation . . . 105
7.1.3 Unfairness . . . 106
7.1.4 Inefficiency . . . 106
7.2 Types of Locks . . . 107
7.2.1 Exclusive Locks . . . 107
7.2.2 Reader-Writer Locks . . . 108
7.2.3 Beyond Reader-Writer Locks . . . 108
7.2.4 Scoped Locking . . . 109
7.3 Locking Implementation Issues . . . 111
7.3.1 Sample Exclusive-Locking Implementation Based on Atomic Exchange . . . 111
7.3.2 Other Exclusive-Locking Implementations . . . 111
7.4 Lock-Based Existence Guarantees . . . 113
7.5 Locking: Hero or Villain? . . . 115
7.5.1 Locking For Applications: Hero! . . . 115
7.5.2 Locking For Parallel Libraries: Just Another Tool . . . 115
7.5.3 Locking For Parallelizing Sequential Libraries: Villain! . . . 117
7.6 Summary . . . 119

www.dbooks.org

8 Data Ownership **121**
8.1 Multiple Processes . . . 121
8.2 Partial Data Ownership and pthreads . . . 122
8.3 Function Shipping . . . 122
8.4 Designated Thread . . . 122
8.5 Privatization . . . 123
8.6 Other Uses of Data Ownership . . . 123

9 Deferred Processing **125**
9.1 Running Example . . . 125
9.2 Reference Counting . . . 126
9.3 Hazard Pointers . . . 128
9.4 Sequence Locks . . . 132
9.5 Read-Copy Update (RCU) . . . 135
9.5.1 Introduction to RCU . . . 135
9.5.2 RCU Fundamentals . . . 140
9.5.3 RCU Linux-Kernel API . . . 146
9.5.4 RCU Usage . . . 155
9.5.5 RCU Related Work . . . 166
9.5.6 RCU Exercises . . . 169
9.6 Which to Choose? . . . 169
9.6.1 Which to Choose? (Overview) . . . 169
9.6.2 Which to Choose? (Details) . . . 170
9.6.3 Which to Choose? (Production Use) . . . 172
9.7 What About Updates? . . . 173

10 Data Structures **175**
10.1 Motivating Application . . . 175
10.2 Partitionable Data Structures . . . 175
10.2.1 Hash-Table Design . . . 176
10.2.2 Hash-Table Implementation . . . 176
10.2.3 Hash-Table Performance . . . 178
10.3 Read-Mostly Data Structures . . . 179
10.3.1 RCU-Protected Hash Table Implementation . . . 179
10.3.2 RCU-Protected Hash Table Performance . . . 180
10.3.3 RCU-Protected Hash Table Discussion . . . 183
10.4 Non-Partitionable Data Structures . . . 183
10.4.1 Resizable Hash Table Design . . . 184
10.4.2 Resizable Hash Table Implementation . . . 185
10.4.3 Resizable Hash Table Discussion . . . 189
10.4.4 Other Resizable Hash Tables . . . 190
10.5 Other Data Structures . . . 192
10.6 Micro-Optimization . . . 192
10.6.1 Specialization . . . 192
10.6.2 Bits and Bytes . . . 193
10.6.3 Hardware Considerations . . . 194
10.7 Summary . . . 195

11 Validation **197**
11.1 Introduction . . . 197
11.1.1 Where Do Bugs Come From? . . . 197
11.1.2 Required Mindset . . . 198
11.1.3 When Should Validation Start? . . . 199
11.1.4 The Open Source Way . . . 201
11.2 Tracing . . . 201
11.3 Assertions . . . 202
11.4 Static Analysis . . . 203
11.5 Code Review . . . 203
11.5.1 Inspection . . . 203
11.5.2 Walkthroughs . . . 204
11.5.3 Self-Inspection . . . 204
11.6 Probability and Heisenbugs . . . 205
11.6.1 Statistics for Discrete Testing . . . 206
11.6.2 Statistics Abuse for Discrete Testing . . . 207
11.6.3 Statistics for Continuous Testing . . . 207
11.6.4 Hunting Heisenbugs . . . 208
11.7 Performance Estimation . . . 211
11.7.1 Benchmarking . . . 212
11.7.2 Profiling . . . 212
11.7.3 Differential Profiling . . . 213
11.7.4 Microbenchmarking . . . 213
11.7.5 Isolation . . . 213
11.7.6 Detecting Interference . . . 214
11.8 Summary . . . 216

12 Formal Verification **219**
12.1 State-Space Search . . . 219
12.1.1 Promela and Spin . . . 219
12.1.2 How to Use Promela . . . 221
12.1.3 Promela Example: Locking . . . 224
12.1.4 Promela Example: QRCU . . . 226
12.1.5 Promela Parable: dynticks and Preemptible RCU . . . 231
12.1.6 Validating Preemptible RCU and dynticks . . . 234
12.2 Special-Purpose State-Space Search . . . 246
12.2.1 Anatomy of a Litmus Test . . . 247
12.2.2 What Does This Litmus Test Mean? . . . 247
12.2.3 Running a Litmus Test . . . 248
12.2.4 PPCMEM Discussion . . . 248
12.3 Axiomatic Approaches . . . 249
12.3.1 Axiomatic Approaches and Locking . . . 250
12.3.2 Axiomatic Approaches and RCU . . . 251
12.4 SAT Solvers . . . 253
12.5 Stateless Model Checkers . . . 254
12.6 Summary . . . 254
12.7 Choosing a Validation Plan . . . 255

www.dbooks.org

13 Putting It All Together **259**
13.1 Counter Conundrums . . . 259
13.1.1 Counting Updates . . . 259
13.1.2 Counting Lookups . . . 259
13.2 Refurbish Reference Counting . . . 260
13.2.1 Implementation of Reference-Counting Categories . . . 261
13.2.2 Counter Optimizations . . . 264
13.3 Hazard-Pointer Helpers . . . 264
13.3.1 Scalable Reference Count . . . 264
13.4 Sequence-Locking Specials . . . 264
13.4.1 Correlated Data Elements . . . 264
13.4.2 Upgrade to Writer . . . 265
13.5 RCU Rescues . . . 265
13.5.1 RCU and Per-Thread-Variable-Based Statistical Counters . . . 265
13.5.2 RCU and Counters for Removable I/O Devices . . . 267
13.5.3 Array and Length . . . 268
13.5.4 Correlated Fields . . . 268
13.5.5 Update-Friendly Traversal . . . 269
13.5.6 Scalable Reference Count Two . . . 269

14 Advanced Synchronization **271**
14.1 Avoiding Locks . . . 271
14.2 Non-Blocking Synchronization . . . 271
14.2.1 Simple NBS . . . 272
14.2.2 Applicability of NBS Benefits . . . 274
14.2.3 NBS Discussion . . . 276
14.3 Parallel Real-Time Computing . . . 276
14.3.1 What is Real-Time Computing? . . . 276
14.3.2 Who Needs Real-Time? . . . 280
14.3.3 Who Needs Parallel Real-Time? . . . 281
14.3.4 Implementing Parallel Real-Time Systems . . . 282
14.3.5 Implementing Parallel Real-Time Operating Systems . . . 282
14.3.6 Implementing Parallel Real-Time Applications . . . 292
14.3.7 Real Time vs. Real Fast: How to Choose? . . . 295

15 Advanced Synchronization: Memory Ordering **297**
15.1 Ordering: Why and How? . . . 297
15.1.1 Why Hardware Misordering? . . . 298
15.1.2 How to Force Ordering? . . . 299
15.1.3 Basic Rules of Thumb . . . 302
15.2 Tricks and Traps . . . 303
15.2.1 Variables With Multiple Values . . . 303
15.2.2 Memory-Reference Reordering . . . 304
15.2.3 Address Dependencies . . . 307
15.2.4 Data Dependencies . . . 308
15.2.5 Control Dependencies . . . 309
15.2.6 Cache Coherence . . . 309
15.2.7 Multicopy Atomicity . . . 310
15.3 Compile-Time Consternation . . . 317
15.3.1 Memory-Reference Restrictions . . . 317

15.3.2 Address- and Data-Dependency Difficulties . . . 318
15.3.3 Control-Dependency Calamities . . . 320
15.4 Higher-Level Primitives . . . 323
15.4.1 Memory Allocation . . . 323
15.4.2 RCU . . . 324
15.5 Hardware Specifics . . . 331
15.5.1 Alpha . . . 334
15.5.2 Armv7-A/R . . . 335
15.5.3 Armv8 . . . 336
15.5.4 Itanium . . . 336
15.5.5 MIPS . . . 337
15.5.6 POWER / PowerPC . . . 337
15.5.7 SPARC TSO . . . 338
15.5.8 x86 . . . 339
15.5.9 z Systems . . . 340
15.6 Where is Memory Ordering Needed? . . . 340

16 Ease of Use 343
16.1 What is Easy? . . . 343
16.2 Rusty Scale for API Design . . . 343
16.3 Shaving the Mandelbrot Set . . . 344

17 Conflicting Visions of the Future 347
17.1 The Future of CPU Technology Ain't What it Used to Be . . . 347
17.1.1 Uniprocessor Über Alles . . . 347
17.1.2 Multithreaded Mania . . . 348
17.1.3 More of the Same . . . 349
17.1.4 Crash Dummies Slamming into the Memory Wall . . . 349
17.1.5 Astounding Accelerators . . . 351
17.2 Transactional Memory . . . 351
17.2.1 Outside World . . . 351
17.2.2 Process Modification . . . 354
17.2.3 Synchronization . . . 358
17.2.4 Discussion . . . 361
17.3 Hardware Transactional Memory . . . 363
17.3.1 HTM Benefits WRT to Locking . . . 363
17.3.2 HTM Weaknesses WRT Locking . . . 364
17.3.3 HTM Weaknesses WRT to Locking When Augmented . . . 368
17.3.4 Where Does HTM Best Fit In? . . . 371
17.3.5 Potential Game Changers . . . 371
17.3.6 Conclusions . . . 375
17.4 Formal Regression Testing? . . . 375
17.4.1 Automatic Translation . . . 375
17.4.2 Environment . . . 376
17.4.3 Overhead . . . 376
17.4.4 Locate Bugs . . . 377
17.4.5 Minimal Scaffolding . . . 378
17.4.6 Relevant Bugs . . . 378
17.4.7 Formal Regression Scorecard . . . 379
17.5 Functional Programming for Parallelism . . . 379

www.dbooks.org

17.6 Summary 381

18 Looking Forward and Back **383**

A Important Questions **387**
A.1 What Does "After" Mean? 387
A.2 What is the Difference Between "Concurrent" and "Parallel"? 389
A.3 What Time Is It? 390
A.4 How Much Ordering? 390
A.4.1 Where is the Defining Data? 391
A.4.2 Consistent Data Used Consistently? 391
A.4.3 Is the Problem Partitionable? 392
A.4.4 None of the Above? 392

B "Toy" RCU Implementations **393**
B.1 Lock-Based RCU 393
B.2 Per-Thread Lock-Based RCU 394
B.3 Simple Counter-Based RCU 394
B.4 Starvation-Free Counter-Based RCU 395
B.5 Scalable Counter-Based RCU 397
B.6 Scalable Counter-Based RCU With Shared Grace Periods 398
B.7 RCU Based on Free-Running Counter 400
B.8 Nestable RCU Based on Free-Running Counter 401
B.9 RCU Based on Quiescent States 403
B.10 Summary of Toy RCU Implementations 404

C Why Memory Barriers? **407**
C.1 Cache Structure 407
C.2 Cache-Coherence Protocols 409
C.2.1 MESI States 409
C.2.2 MESI Protocol Messages 409
C.2.3 MESI State Diagram 410
C.2.4 MESI Protocol Example 411
C.3 Stores Result in Unnecessary Stalls 411
C.3.1 Store Buffers 412
C.3.2 Store Forwarding 412
C.3.3 Store Buffers and Memory Barriers 413
C.4 Store Sequences Result in Unnecessary Stalls 415
C.4.1 Invalidate Queues 415
C.4.2 Invalidate Queues and Invalidate Acknowledge 415
C.4.3 Invalidate Queues and Memory Barriers 416
C.5 Read and Write Memory Barriers 417
C.6 Example Memory-Barrier Sequences 418
C.6.1 Ordering-Hostile Architecture 418
C.6.2 Example 1 419
C.6.3 Example 2 419
C.6.4 Example 3 419
C.7 Are Memory Barriers Forever? 420
C.8 Advice to Hardware Designers 420

D Style Guide **423**
D.1 Paul's Conventions . . . 423
D.2 NIST Style Guide . . . 424
D.2.1 Unit Symbol . . . 424
D.2.2 NIST Guide Yet To Be Followed . . . 425
D.3 LaTeX Conventions . . . 425
D.3.1 Monospace Font . . . 425
D.3.2 Cross-reference . . . 429
D.3.3 Non Breakable Spaces . . . 430
D.3.4 Hyphenation and Dashes . . . 430
D.3.5 Punctuation . . . 431
D.3.6 Floating Object Format . . . 431
D.3.7 Improvement Candidates . . . 432

E Answers to Quick Quizzes **437**
E.1 How To Use This Book . . . 437
E.2 Introduction . . . 438
E.3 Hardware and its Habits . . . 441
E.4 Tools of the Trade . . . 445
E.5 Counting . . . 451
E.6 Partitioning and Synchronization Design . . . 464
E.7 Locking . . . 468
E.8 Data Ownership . . . 475
E.9 Deferred Processing . . . 476
E.10 Data Structures . . . 489
E.11 Validation . . . 494
E.12 Formal Verification . . . 500
E.13 Putting It All Together . . . 507
E.14 Advanced Synchronization . . . 509
E.15 Advanced Synchronization: Memory Ordering . . . 511
E.16 Ease of Use . . . 520
E.17 Conflicting Visions of the Future . . . 521
E.18 Important Questions . . . 526
E.19 "Toy" RCU Implementations . . . 527
E.20 Why Memory Barriers? . . . 532

Glossary **537**

Bibliography **545**

Credits **587**
LaTeX Advisor . . . 587
Reviewers . . . 587
Machine Owners . . . 587
Original Publications . . . 587
Figure Credits . . . 588
Other Support . . . 589

www.dbooks.org

If you would only recognize that life is hard, things would be so much easier for you.

Louis D. Brandeis

Chapter 1

How To Use This Book

The purpose of this book is to help you program shared-memory parallel systems without risking your sanity.[1] Nevertheless, you should think of the information in this book as a foundation on which to build, rather than as a completed cathedral. Your mission, if you choose to accept, is to help make further progress in the exciting field of parallel programming—progress that will in time render this book obsolete.

Parallel programming in the 21st century is no longer focused solely on science, research, and grand-challenge projects. And this is all to the good, because it means that parallel programming is becoming an engineering discipline. Therefore, as befits an engineering discipline, this book examines specific parallel-programming tasks and describes how to approach them. In some surprisingly common cases, these tasks can be automated.

This book is written in the hope that presenting the engineering discipline underlying successful parallel-programming projects will free a new generation of parallel hackers from the need to slowly and painstakingly reinvent old wheels, enabling them to instead focus their energy and creativity on new frontiers. However, what you get from this book will be determined by what you put into it. It is hoped that simply reading this book will be helpful, and that working the Quick Quizzes will be even more helpful. However, the best results come from applying the techniques taught in this book to real-life problems. As always, practice makes perfect.

But no matter how you approach it, we sincerely hope that parallel programming brings you at least as much fun, excitement, and challenge that it has brought to us!

[1] Or, perhaps more accurately, without much greater risk to your sanity than that incurred by non-parallel programming. Which, come to think of it, might not be saying all that much.

1.1 Roadmap

Cat: Where are you going?
Alice: Which way should I go?
Cat: That depends on where you are going.
Alice: I don't know.
Cat: Then it doesn't matter which way you go.

Lewis Carroll, Alice in Wonderland

This book is a handbook of widely applicable and heavily used design techniques, rather than a collection of optimal algorithms with tiny areas of applicability. You are currently reading Chapter 1, but you knew that already. Chapter 2 gives a high-level overview of parallel programming.

Chapter 3 introduces shared-memory parallel hardware. After all, it is difficult to write good parallel code unless you understand the underlying hardware. Because hardware constantly evolves, this chapter will always be out of date. We will nevertheless do our best to keep up. Chapter 4 then provides a very brief overview of common shared-memory parallel-programming primitives.

Chapter 5 takes an in-depth look at parallelizing one of the simplest problems imaginable, namely counting. Because almost everyone has an excellent grasp of counting, this chapter is able to delve into many important parallel-programming issues without the distractions of more-typical computer-science problems. My impression is that this chapter has seen the greatest use in parallel-programming coursework.

Chapter 6 introduces a number of design-level methods of addressing the issues identified in Chapter 5. It turns out that it is important to address parallelism at the design level when feasible: To paraphrase Dijkstra [Dij68], "retrofitted parallelism considered grossly suboptimal" [McK12c].

www.dbooks.org

The next three chapters examine three important approaches to synchronization. Chapter 7 covers locking, which is still not only the workhorse of production-quality parallel programming, but is also widely considered to be parallel programming's worst villain. Chapter 8 gives a brief overview of data ownership, an often overlooked but remarkably pervasive and powerful approach. Finally, Chapter 9 introduces a number of deferred-processing mechanisms, including reference counting, hazard pointers, sequence locking, and RCU.

Chapter 10 applies the lessons of previous chapters to hash tables, which are heavily used due to their excellent partitionability, which (usually) leads to excellent performance and scalability.

As many have learned to their sorrow, parallel programming without validation is a sure path to abject failure. Chapter 11 covers various forms of testing. It is of course impossible to test reliability into your program after the fact, so Chapter 12 follows up with a brief overview of a couple of practical approaches to formal verification.

Chapter 13 contains a series of moderate-sized parallel programming problems. The difficulty of these problems vary, but should be appropriate for someone who has mastered the material in the previous chapters.

Chapter 14 looks at advanced synchronization methods, including non-blocking synchronization and parallel real-time computing, while Chapter 15 covers the advanced topic of memory ordering. Chapter 16 follows up with some ease-of-use advice. Chapter 17 looks at a few possible future directions, including shared-memory parallel system design, software and hardware transactional memory, and functional programming for parallelism. Finally, Chapter 18 reviews the material in this book and its origins.

This chapter is followed by a number of appendices. The most popular of these appears to be Appendix C, which delves even further into memory ordering. Appendix E contains the answers to the infamous Quick Quizzes, which are discussed in the next section.

1.2 Quick Quizzes

> Undertake something difficult, otherwise you will never grow.
>
> *Abbreviated from Ronald E. Osburn*

"Quick quizzes" appear throughout this book, and the answers may be found in Appendix E starting on page 437. Some of them are based on material in which that quick quiz appears, but others require you to think beyond that section, and, in some cases, beyond the realm of current knowledge. As with most endeavors, what you get out of this book is largely determined by what you are willing to put into it. Therefore, readers who make a genuine effort to solve a quiz before looking at the answer find their effort repaid handsomely with increased understanding of parallel programming.

Quick Quiz 1.1: Where are the answers to the Quick Quizzes found? ■

Quick Quiz 1.2: Some of the Quick Quiz questions seem to be from the viewpoint of the reader rather than the author. Is that really the intent? ■

Quick Quiz 1.3: These Quick Quizzes are just not my cup of tea. What can I do about it? ■

In short, if you need a deep understanding of the material, then you should invest some time into answering the Quick Quizzes. Don't get me wrong, passively reading the material can be quite valuable, but gaining full problem-solving capability really does require that you practice solving problems.

I learned this the hard way during coursework for my late-in-life Ph.D. I was studying a familiar topic, and was surprised at how few of the chapter's exercises I could answer off the top of my head.[2] Forcing myself to answer the questions greatly increased my retention of the material. So with these Quick Quizzes I am not asking you to do anything that I have not been doing myself.

Finally, the most common learning disability is thinking that you already understand the material at hand. The quick quizzes can be an extremely effective cure.

1.3 Alternatives to This Book

> Between two evils I always pick the one I never tried before.
>
> *Mae West*

As Knuth learned the hard way, if you want your book to be finite, it must be focused. This book focuses on shared-memory parallel programming, with an emphasis on software that lives near the bottom of the software

[2] So I suppose that it was just as well that my professors refused to let me waive that class!

stack, such as operating-system kernels, parallel data-management systems, low-level libraries, and the like. The programming language used by this book is C.

If you are interested in other aspects of parallelism, you might well be better served by some other book. Fortunately, there are many alternatives available to you:

1. If you prefer a more academic and rigorous treatment of parallel programming, you might like Herlihy's and Shavit's textbook [HS08, HSLS20]. This book starts with an interesting combination of low-level primitives at high levels of abstraction from the hardware, and works its way through locking and simple data structures including lists, queues, hash tables, and counters, culminating with transactional memory. Michael Scott's textbook [Sco13] approaches similar material with more of a software-engineering focus, and, as far as I know, is the first formally published academic textbook with section devoted to RCU.

2. If you would like an academic treatment of parallel programming from a programming-language-pragmatics viewpoint, you might be interested in the concurrency chapter from Scott's textbook [Sco06, Sco15] on programming-language pragmatics.

3. If you are interested in an object-oriented patternist treatment of parallel programming focussing on C++, you might try Volumes 2 and 4 of Schmidt's POSA series [SSRB00, BHS07]. Volume 4 in particular has some interesting chapters applying this work to a warehouse application. The realism of this example is attested to by the section entitled "Partitioning the Big Ball of Mud", in which the problems inherent in parallelism often take a back seat to getting one's head around a real-world application.

4. If you want to work with Linux-kernel device drivers, then Corbet's, Rubini's, and Kroah-Hartman's "Linux Device Drivers" [CRKH05] is indispensable, as is the Linux Weekly News web site (`https://lwn.net/`). There is a large number of books and resources on the more general topic of Linux kernel internals.

5. If your primary focus is scientific and technical computing, and you prefer a patternist approach, you might try Mattson et al.'s textbook [MSM05]. It covers Java, C/C++, OpenMP, and MPI. Its patterns are admirably focused first on design, then on implementation.

6. If your primary focus is scientific and technical computing, and you are interested in GPUs, CUDA, and MPI, you might check out Norm Matloff's "Programming on Parallel Machines" [Mat17]. Of course, the GPU vendors have quite a bit of additional information [AMD20, Zel11, NVi17a, NVi17b].

7. If you are interested in POSIX Threads, you might take a look at David R. Butenhof's book [But97]. In addition, W. Richard Stevens's book [Ste92, Ste13] covers UNIX and POSIX, and Stewart Weiss's lecture notes [Wei13] provide an thorough and accessible introduction with a good set of examples.

8. If you are interested in C++11, you might like Anthony Williams's "C++ Concurrency in Action: Practical Multithreading" [Wil12, Wil19].

9. If you are interested in C++, but in a Windows environment, you might try Herb Sutter's "Effective Concurrency" series in Dr. Dobbs Journal [Sut08]. This series does a reasonable job of presenting a commonsense approach to parallelism.

10. If you want to try out Intel Threading Building Blocks, then perhaps James Reinders's book [Rei07] is what you are looking for.

11. Those interested in learning how various types of multi-processor hardware cache organizations affect the implementation of kernel internals should take a look at Curt Schimmel's classic treatment of this subject [Sch94].

12. If you are looking for a hardware view, Hennessy's and Patterson's classic textbook [HP17, HP11] is well worth a read. If you are looking for an academic textbook on memory ordering, that of Daniel Sorin et al. [SHW11, NSHW20] is highly recommended. For a memory-ordering tutorial from a Linux-kernel viepwpoint, Paolo Bonzini's LWN series is a good place to start [Bon21a, Bon21d, Bon21c, Bon21b].

13. Finally, those using Java might be well-served by Doug Lea's textbooks [Lea97, GPB+07].

However, if you are interested in principles of parallel design for low-level software, especially software written in C, read on!

www.dbooks.org

1.4 Sample Source Code

> Use the source, Luke!
>
> *Unknown Star Wars fan*

This book discusses its fair share of source code, and in many cases this source code may be found in the `CodeSamples` directory of this book's git tree. For example, on UNIX systems, you should be able to type the following:

```
find CodeSamples -name rcu_rcpls.c -print
```

This command will locate the file `rcu_rcpls.c`, which is called out in Appendix B. Other types of systems have well-known ways of locating files by filename.

1.5 Whose Book Is This?

> If you become a teacher, by your pupils you'll be taught.
>
> *Oscar Hammerstein II*

As the cover says, the editor is one Paul E. McKenney. However, the editor does accept contributions via the `perfbook@vger.kernel.org` email list. These contributions can be in pretty much any form, with popular approaches including text emails, patches against the book's LaTeX source, and even `git pull` requests. Use whatever form works best for you.

To create patches or `git pull` requests, you will need the LaTeX source to the book, which is at `git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git`. You will of course also need `git` and LaTeX, which are available as part of most mainstream Linux distributions. Other packages may be required, depending on the distribution you use. The required list of packages for a few popular distributions is listed in the file `FAQ-BUILD.txt` in the LaTeX source to the book.

To create and display a current LaTeX source tree of this book, use the list of Linux commands shown in Listing 1.1. In some environments, the `evince` command that displays `perfbook.pdf` may need to be replaced, for example, with `acroread`. The `git clone` command need only be used the first time you create a PDF, subsequently, you can run the commands shown in Listing 1.2 to pull in any updates and generate an updated PDF. The commands in Listing 1.2 must be run within the `perfbook` directory created by the commands shown in Listing 1.1.

Listing 1.1: Creating an Up-To-Date PDF

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/↵
↪  paulmck/perfbook.git
cd perfbook
# You may need to install a font. See item 1 in FAQ.txt.
make                    # -jN for parallel build
evince perfbook.pdf &   # Two-column version
make perfbook-1c.pdf
evince perfbook-1c.pdf & # One-column version for e-readers
make help               # Display other build options
```

Listing 1.2: Generating an Updated PDF

```
git remote update
git checkout origin/master
make                    # -jN for parallel build
evince perfbook.pdf &   # Two-column version
make perfbook-1c.pdf
evince perfbook-1c.pdf & # One-column version for e-readers
```

PDFs of this book are sporadically posted at `https://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html` and at `http://www.rdrop.com/users/paulmck/perfbook/`.

The actual process of contributing patches and sending `git pull` requests is similar to that of the Linux kernel, which is documented in the `Documentation/SubmittingPatches` file in the Linux source tree. One important requirement is that each patch (or commit, in the case of a `git pull` request) must contain a valid `Signed-off-by:` line, which has the following format:

```
Signed-off-by: My Name <myname@example.org>
```

Please see `https://lkml.org/lkml/2007/1/15/219` for an example patch containing a `Signed-off-by:` line.

It is important to note that the `Signed-off-by:` line has a very specific meaning, namely that you are certifying that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source License and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to

submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

This is quite similar to the Developer's Certificate of Origin (DCO) 1.1 used by the Linux kernel. You must use your real name: I unfortunately cannot accept pseudonymous or anonymous contributions.

The language of this book is American English, however, the open-source nature of this book permits translations, and I personally encourage them. The open-source licenses covering this book additionally allow you to sell your translation, if you wish. I do request that you send me a copy of the translation (hardcopy if available), but this is a request made as a professional courtesy, and is not in any way a prerequisite to the permission that you already have under the Creative Commons and GPL licenses. Please see the `FAQ.txt` file in the source tree for a list of translations currently in progress. I consider a translation effort to be "in progress" once at least one chapter has been fully translated.

There are many styles under the "American English" rubric. The style for this particular book is documented in Appendix D.

As noted at the beginning of this section, I am this book's editor. However, if you choose to contribute, it will be your book as well. In that spirit, I offer you Chapter 2, our introduction.

www.dbooks.org

Chapter 2

Introduction

> If parallel programming is so hard, why are there so many parallel programs?
>
> *Unknown*

Parallel programming has earned a reputation as one of the most difficult areas a hacker can tackle. Papers and textbooks warn of the perils of deadlock, livelock, race conditions, non-determinism, Amdahl's-Law limits to scaling, and excessive realtime latencies. And these perils are quite real; we authors have accumulated uncounted years of experience along with the resulting emotional scars, grey hairs, and hair loss.

However, new technologies that are difficult to use at introduction invariably become easier over time. For example, the once-rare ability to drive a car is now commonplace in many countries. This dramatic change came about for two basic reasons: (1) cars became cheaper and more readily available, so that more people had the opportunity to learn to drive, and (2) cars became easier to operate due to automatic transmissions, automatic chokes, automatic starters, greatly improved reliability, and a host of other technological improvements.

The same is true for many other technologies, including computers. It is no longer necessary to operate a keypunch in order to program. Spreadsheets allow most non-programmers to get results from their computers that would have required a team of specialists a few decades ago. Perhaps the most compelling example is web-surfing and content creation, which since the early 2000s has been easily done by untrained, uneducated people using various now-commonplace social-networking tools. As recently as 1968, such content creation was a far-out research project [Eng68], described at the time as "like a UFO landing on the White House lawn"[Gri00].

Therefore, if you wish to argue that parallel programming will remain as difficult as it is currently perceived by many to be, it is you who bears the burden of proof, keeping in mind the many centuries of counter-examples in many fields of endeavor.

2.1 Historic Parallel Programming Difficulties

> Not the power to remember, but its very opposite, the power to forget, is a necessary condition for our existence.
>
> *Sholem Asch*

As indicated by its title, this book takes a different approach. Rather than complain about the difficulty of parallel programming, it instead examines the reasons why parallel programming is difficult, and then works to help the reader to overcome these difficulties. As will be seen, these difficulties have historically fallen into several categories, including:

1. The historic high cost and relative rarity of parallel systems.
2. The typical researcher's and practitioner's lack of experience with parallel systems.
3. The paucity of publicly accessible parallel code.
4. The lack of a widely understood engineering discipline of parallel programming.
5. The high overhead of communication relative to that of processing, even in tightly coupled shared-memory computers.

Many of these historic difficulties are well on the way to being overcome. First, over the past few decades, the cost of parallel systems has decreased from many multiples of that of a house to that of a modest meal, courtesy of Moore's Law. Papers calling out the advantages of multicore CPUs were published as early as

www.dbooks.org

1996 [ONH+96]. IBM introduced simultaneous multithreading into its high-end POWER family in 2000, and multicore in 2001. Intel introduced hyperthreading into its commodity Pentium line in November 2000, and both AMD and Intel introduced dual-core CPUs in 2005. Sun followed with the multicore/multi-threaded Niagara in late 2005. In fact, by 2008, it was becoming difficult to find a single-CPU desktop system, with single-core CPUs being relegated to netbooks and embedded devices. By 2012, even smartphones were starting to sport multiple CPUs. By 2020, safety-critical software standards started addressing concurrency.

Second, the advent of low-cost and readily available multicore systems means that the once-rare experience of parallel programming is now available to almost all researchers and practitioners. In fact, parallel systems have long been within the budget of students and hobbyists. We can therefore expect greatly increased levels of invention and innovation surrounding parallel systems, and that increased familiarity will over time make the once prohibitively expensive field of parallel programming much more friendly and commonplace.

Third, in the 20th century, large systems of highly parallel software were almost always closely guarded proprietary secrets. In happy contrast, the 21st century has seen numerous open-source (and thus publicly available) parallel software projects, including the Linux kernel [Tor03], database systems [Pos08, MS08], and message-passing systems [The08, Uni08a]. This book will draw primarily from the Linux kernel, but will provide much material suitable for user-level applications.

Fourth, even though the large-scale parallel-programming projects of the 1980s and 1990s were almost all proprietary projects, these projects have seeded other communities with cadres of developers who understand the engineering discipline required to develop production-quality parallel code. A major purpose of this book is to present this engineering discipline.

Unfortunately, the fifth difficulty, the high cost of communication relative to that of processing, remains largely in force. This difficulty has been receiving increasing attention during the new millennium. However, according to Stephen Hawking, the finite speed of light and the atomic nature of matter will limit progress in this area [Gar07, Moo03]. Fortunately, this difficulty has been in force since the late 1980s, so that the aforementioned engineering discipline has evolved practical and effective strategies for handling it. In addition, hardware designers are increasingly aware of these issues, so perhaps future hardware will be more friendly to parallel software, as discussed in Section 3.3.

Quick Quiz 2.1: Come on now!!! Parallel programming has been known to be exceedingly hard for many decades. You seem to be hinting that it is not so hard. What sort of game are you playing? ■

However, even though parallel programming might not be as hard as is commonly advertised, it is often more work than is sequential programming.

Quick Quiz 2.2: How could parallel programming *ever* be as easy as sequential programming? ■

It therefore makes sense to consider alternatives to parallel programming. However, it is not possible to reasonably consider parallel-programming alternatives without understanding parallel-programming goals. This topic is addressed in the next section.

2.2 Parallel Programming Goals

> If you don't know where you are going, you will end up somewhere else.
>
> *Yogi Berra*

The three major goals of parallel programming (over and above those of sequential programming) are as follows:

1. Performance.
2. Productivity.
3. Generality.

Unfortunately, given the current state of the art, it is possible to achieve at best two of these three goals for any given parallel program. These three goals therefore form the *iron triangle of parallel programming*, a triangle upon which overly optimistic hopes all too often come to grief.[1]

Quick Quiz 2.3: Oh, really??? What about correctness, maintainability, robustness, and so on? ■

Quick Quiz 2.4: And if correctness, maintainability, and robustness don't make the list, why do productivity and generality? ■

[1] Kudos to Michael Wong for naming the iron triangle.

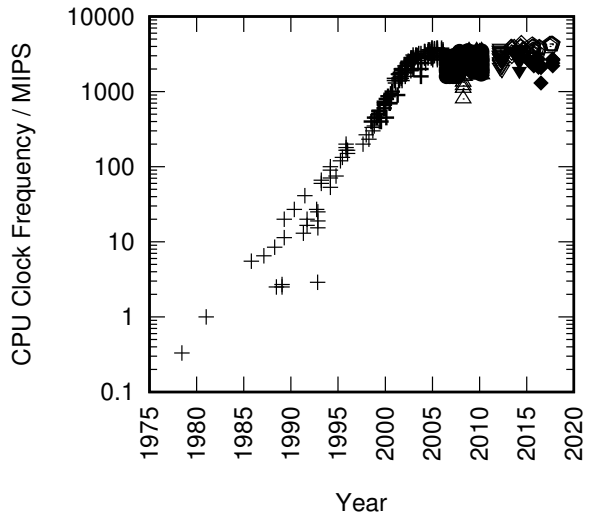


Figure 2.1: MIPS/Clock-Frequency Trend for Intel CPUs

Quick Quiz 2.5: Given that parallel programs are much harder to prove correct than are sequential programs, again, shouldn't correctness *really* be on the list? ■

Quick Quiz 2.6: What about just having fun? ■

Each of these goals is elaborated upon in the following sections.

2.2.1 Performance

Performance is the primary goal behind most parallel-programming effort. After all, if performance is not a concern, why not do yourself a favor: Just write sequential code, and be happy? It will very likely be easier and you will probably get done much more quickly.

Quick Quiz 2.7: Are there no cases where parallel programming is about something other than performance? ■

Note that "performance" is interpreted broadly here, including for example scalability (performance per CPU) and efficiency (performance per watt).

That said, the focus of performance has shifted from hardware to parallel software. This change in focus is due to the fact that, although Moore's Law continues to deliver increases in transistor density, it has ceased to provide the traditional single-threaded performance increases. This can be seen in Figure 2.1[2], which shows that writing single-threaded code and simply waiting a year or two for the CPUs to catch up may no longer be an option. Given the recent trends on the part of all major manufacturers towards multicore/multithreaded systems, parallelism is the way to go for those wanting to avail themselves of the full performance of their systems.

Quick Quiz 2.8: Why not instead rewrite programs from inefficient scripting languages to C or C++? ■

Even so, the first goal is performance rather than scalability, especially given that the easiest way to attain linear scalability is to reduce the performance of each CPU [Tor01]. Given a four-CPU system, which would you prefer? A program that provides 100 transactions per second on a single CPU, but does not scale at all? Or a program that provides 10 transactions per second on a single CPU, but scales perfectly? The first program seems like a better bet, though the answer might change if you happened to have a 32-CPU system.

That said, just because you have multiple CPUs is not necessarily in and of itself a reason to use them all, especially given the recent decreases in price of multi-CPU systems. The key point to understand is that parallel programming is primarily a performance optimization, and, as such, it is one potential optimization of many. If your program is fast enough as currently written, there is no reason to optimize, either by parallelizing it or by applying any of a number of potential sequential optimizations.[3] By the same token, if you are looking to apply parallelism as an optimization to a sequential program, then you will need to compare parallel algorithms to the best sequential algorithms. This may require some care, as far too many publications ignore the sequential case when analyzing the performance of parallel algorithms.

[2] This plot shows clock frequencies for newer CPUs theoretically capable of retiring one or more instructions per clock, and MIPS (millions of instructions per second, usually from the old Dhrystone benchmark) for older CPUs requiring multiple clocks to execute even the simplest instruction. The reason for shifting between these two measures is that the newer CPUs' ability to retire multiple instructions per clock is typically limited by memory-system performance. Furthermore, the benchmarks commonly used on the older CPUs are obsolete, and it is difficult to run the newer benchmarks on systems containing the old CPUs, in part because it is hard to find working instances of the old CPUs.

[3] Of course, if you are a hobbyist whose primary interest is writing parallel software, that is more than enough reason to parallelize whatever software you are interested in.

www.dbooks.org

2.2.2 Productivity

Quick Quiz 2.9: Why all this prattling on about non-technical issues??? And not just *any* non-technical issue, but *productivity* of all things? Who cares? ■

Productivity has been becoming increasingly important in recent decades. To see this, consider that the price of early computers was tens of millions of dollars at a time when engineering salaries were but a few thousand dollars a year. If dedicating a team of ten engineers to such a machine would improve its performance, even by only 10 %, then their salaries would be repaid many times over.

One such machine was the CSIRAC, the oldest still-intact stored-program computer, which was put into operation in 1949 [Mus04, Dep06]. Because this machine was built before the transistor era, it was constructed of 2,000 vacuum tubes, ran with a clock frequency of 1 kHz, consumed 30 kW of power, and weighed more than three metric tons. Given that this machine had but 768 words of RAM, it is safe to say that it did not suffer from the productivity issues that often plague today's large-scale software projects.

Today, it would be quite difficult to purchase a machine with so little computing power. Perhaps the closest equivalents are 8-bit embedded microprocessors exemplified by the venerable Z80 [Wik08], but even the old Z80 had a CPU clock frequency more than 1,000 times faster than the CSIRAC. The Z80 CPU had 8,500 transistors, and could be purchased in 2008 for less than $2 US per unit in 1,000-unit quantities. In stark contrast to the CSIRAC, software-development costs are anything but insignificant for the Z80.

The CSIRAC and the Z80 are two points in a long-term trend, as can be seen in Figure 2.2. This figure plots an approximation to computational power per die over the past four decades, showing an impressive six-order-of-magnitude increase over a period of forty years. Note that the advent of multicore CPUs has permitted this increase to continue apace despite the clock-frequency wall encountered in 2003, albeit courtesy of dies supporting more than 50 hardware threads each.

One of the inescapable consequences of the rapid decrease in the cost of hardware is that software productivity becomes increasingly important. It is no longer sufficient merely to make efficient use of the hardware: It is now necessary to make extremely efficient use of software developers as well. This has long been the case for sequential hardware, but parallel hardware has become a low-cost commodity only recently. Therefore, only recently has high productivity become critically important when creating parallel software.

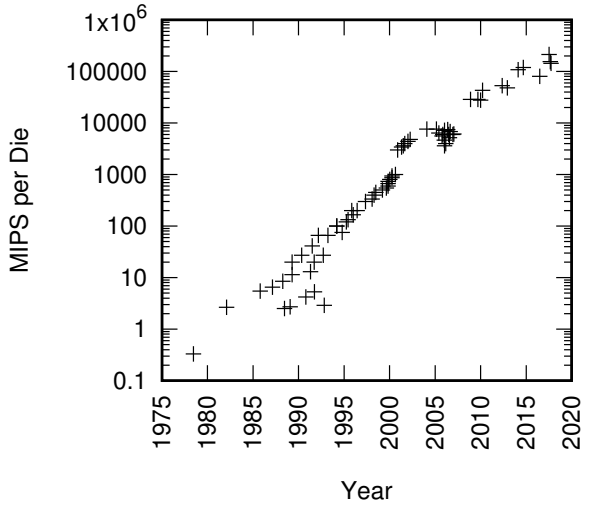


Figure 2.2: MIPS per Die for Intel CPUs

Quick Quiz 2.10: Given how cheap parallel systems have become, how can anyone afford to pay people to program them? ■

Perhaps at one time, the sole purpose of parallel software was performance. Now, however, productivity is gaining the spotlight.

2.2.3 Generality

One way to justify the high cost of developing parallel software is to strive for maximal generality. All else being equal, the cost of a more-general software artifact can be spread over more users than that of a less-general one. In fact, this economic force explains much of the maniacal focus on portability, which can be seen as an important special case of generality.[4]

Unfortunately, generality often comes at the cost of performance, productivity, or both. For example, portability is often achieved via adaptation layers, which inevitably exact a performance penalty. To see this more generally, consider the following popular parallel programming environments:

C/C++ "Locking Plus Threads": This category, which includes POSIX Threads (pthreads) [Ope97], Windows Threads, and numerous operating-system kernel environments, offers excellent performance (at

[4] Kudos to Michael Wong for pointing this out.

least within the confines of a single SMP system) and also offers good generality. Pity about the relatively low productivity.

Java: This general purpose and inherently multithreaded programming environment is widely believed to offer much higher productivity than C or C++, courtesy of the automatic garbage collector and the rich set of class libraries. However, its performance, though greatly improved in the early 2000s, lags that of C and C++.

MPI: This Message Passing Interface [MPI08] powers the largest scientific and technical computing clusters in the world and offers unparalleled performance and scalability. In theory, it is general purpose, but it is mainly used for scientific and technical computing. Its productivity is believed by many to be even lower than that of C/C++ "locking plus threads" environments.

OpenMP: This set of compiler directives can be used to parallelize loops. It is thus quite specific to this task, and this specificity often limits its performance. It is, however, much easier to use than MPI or C/C++ "locking plus threads."

SQL: Structured Query Language [Int92] is specific to relational database queries. However, its performance is quite good as measured by the Transaction Processing Performance Council (TPC) benchmark results [Tra01]. Productivity is excellent; in fact, this parallel programming environment enables people to make good use of a large parallel system despite having little or no knowledge of parallel programming concepts.

The nirvana of parallel programming environments, one that offers world-class performance, productivity, and generality, simply does not yet exist. Until such a nirvana appears, it will be necessary to make engineering tradeoffs among performance, productivity, and generality. One such tradeoff is shown in Figure 2.3, which shows how productivity becomes increasingly important at the upper layers of the system stack, while performance and generality become increasingly important at the lower layers of the system stack. The huge development costs incurred at the lower layers must be spread over equally huge numbers of users (hence the importance of generality), and performance lost in lower layers cannot easily be recovered further up the stack. In the upper layers of the stack, there

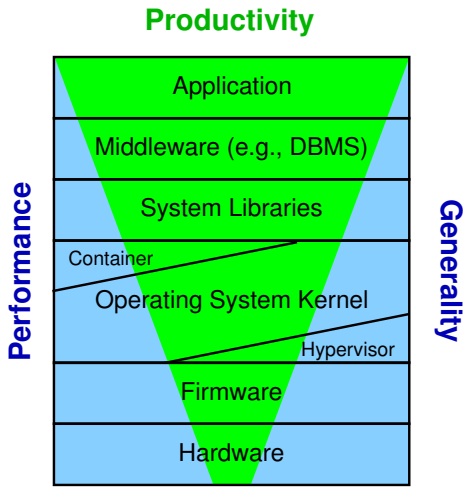


Figure 2.3: Software Layers and Performance, Productivity, and Generality

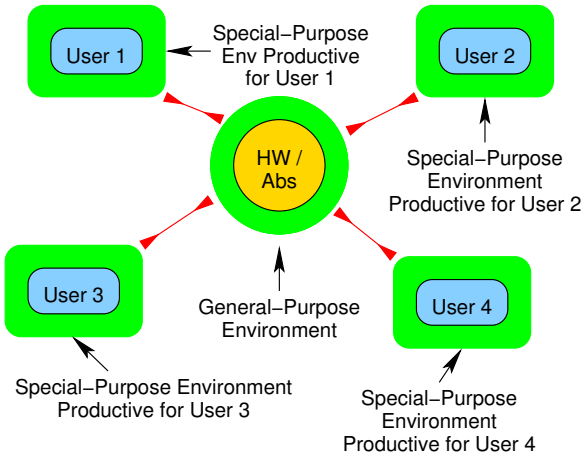


Figure 2.4: Tradeoff Between Productivity and Generality

might be very few users for a given specific application, in which case productivity concerns are paramount. This explains the tendency towards "bloatware" further up the stack: Extra hardware is often cheaper than extra developers. This book is intended for developers working near the bottom of the stack, where performance and generality are of greatest concern.

It is important to note that a tradeoff between productivity and generality has existed for centuries in many fields. For but one example, a nailgun is more productive than a hammer for driving nails, but in contrast to the nailgun, a hammer can be used for many things besides driving nails. It should therefore be no surprise to see

www.dbooks.org

similar tradeoffs appear in the field of parallel computing. This tradeoff is shown schematically in Figure 2.4. Here, users 1, 2, 3, and 4 have specific jobs that they need the computer to help them with. The most productive possible language or environment for a given user is one that simply does that user's job, without requiring any programming, configuration, or other setup.

Quick Quiz 2.11: This is a ridiculously unachievable ideal! Why not focus on something that is achievable in practice? ■

Unfortunately, a system that does the job required by user 1 is unlikely to do user 2's job. In other words, the most productive languages and environments are domain-specific, and thus by definition lacking generality.

Another option is to tailor a given programming language or environment to the hardware system (for example, low-level languages such as assembly, C, C++, or Java) or to some abstraction (for example, Haskell, Prolog, or Snobol), as is shown by the circular region near the center of Figure 2.4. These languages can be considered to be general in the sense that they are equally ill-suited to the jobs required by users 1, 2, 3, and 4. In other words, their generality comes at the expense of decreased productivity when compared to domain-specific languages and environments. Worse yet, a language that is tailored to a given abstraction is likely to suffer from performance and scalability problems unless and until it can be efficiently mapped to real hardware.

Is there no escape from iron triangle's three conflicting goals of performance, productivity, and generality?

It turns out that there often is an escape, for example, using the alternatives to parallel programming discussed in the next section. After all, parallel programming can be a great deal of fun, but it is not always the best tool for the job.

2.3 Alternatives to Parallel Programming

> Experiment is folly when experience shows the way.
>
> *Roger M. Babson*

In order to properly consider alternatives to parallel programming, you must first decide on what exactly you expect the parallelism to do for you. As seen in Section 2.2, the primary goals of parallel programming are performance, productivity, and generality. Because this book is intended for developers working on performance-critical code near the bottom of the software stack, the remainder of this section focuses primarily on performance improvement.

It is important to keep in mind that parallelism is but one way to improve performance. Other well-known approaches include the following, in roughly increasing order of difficulty:

1. Run multiple instances of a sequential application.
2. Make the application use existing parallel software.
3. Optimize the serial application.

These approaches are covered in the following sections.

2.3.1 Multiple Instances of a Sequential Application

Running multiple instances of a sequential application can allow you to do parallel programming without actually doing parallel programming. There are a large number of ways to approach this, depending on the structure of the application.

If your program is analyzing a large number of different scenarios, or is analyzing a large number of independent data sets, one easy and effective approach is to create a single sequential program that carries out a single analysis, then use any of a number of scripting environments (for example the `bash` shell) to run a number of instances of that sequential program in parallel. In some cases, this approach can be easily extended to a cluster of machines.

This approach may seem like cheating, and in fact some denigrate such programs as "embarrassingly parallel". And in fact, this approach does have some potential disadvantages, including increased memory consumption, waste of CPU cycles recomputing common intermediate results, and increased copying of data. However, it is often extremely productive, garnering extreme performance gains with little or no added effort.

2.3.2 Use Existing Parallel Software

There is no longer any shortage of parallel software environments that can present a single-threaded programming environment, including relational databases [Dat82], web-application servers, and map-reduce environments. For example, a common design provides a separate process for each user, each of which generates SQL from user queries. This per-user SQL is run against a common relational

database, which automatically runs the users' queries concurrently. The per-user programs are responsible only for the user interface, with the relational database taking full responsibility for the difficult issues surrounding parallelism and persistence.

In addition, there are a growing number of parallel library functions, particularly for numeric computation. Even better, some libraries take advantage of special-purpose hardware such as vector units and general-purpose graphical processing units (GPGPUs).

Taking this approach often sacrifices some performance, at least when compared to carefully hand-coding a fully parallel application. However, such sacrifice is often well repaid by a huge reduction in development effort.

Quick Quiz 2.12: Wait a minute! Doesn't this approach simply shift the development effort from you to whoever wrote the existing parallel software you are using? ■

2.3.3 Performance Optimization

Up through the early 2000s, CPU clock frequencies doubled every 18 months. It was therefore usually more important to create new functionality than to do careful performance optimization. Now that Moore's Law is "only" increasing transistor density instead of increasing both transistor density and per-transistor performance, it might be a good time to rethink the importance of performance optimization. After all, new hardware generations no longer bring significant single-threaded performance improvements. Furthermore, many performance optimizations can also conserve energy.

From this viewpoint, parallel programming is but another performance optimization, albeit one that is becoming much more attractive as parallel systems become cheaper and more readily available. However, it is wise to keep in mind that the speedup available from parallelism is limited to roughly the number of CPUs (but see Section 6.5 for an interesting exception). In contrast, the speedup available from traditional single-threaded software optimizations can be much larger. For example, replacing a long linked list with a hash table or a search tree can improve performance by many orders of magnitude. This highly optimized single-threaded program might run much faster than its unoptimized parallel counterpart, making parallelization unnecessary. Of course, a highly optimized parallel program would be even better, aside from the added development effort required.

Furthermore, different programs might have different performance bottlenecks. For example, if your program spends most of its time waiting on data from your disk drive, using multiple CPUs will probably just increase the time wasted waiting for the disks. In fact, if the program was reading from a single large file laid out sequentially on a rotating disk, parallelizing your program might well make it a lot slower due to the added seek overhead. You should instead optimize the data layout so that the file can be smaller (thus faster to read), split the file into chunks which can be accessed in parallel from different drives, cache frequently accessed data in main memory, or, if possible, reduce the amount of data that must be read.

Quick Quiz 2.13: What other bottlenecks might prevent additional CPUs from providing additional performance? ■

Parallelism can be a powerful optimization technique, but it is not the only such technique, nor is it appropriate for all situations. Of course, the easier it is to parallelize your program, the more attractive parallelization becomes as an optimization. Parallelization has a reputation of being quite difficult, which leads to the question "exactly what makes parallel programming so difficult?"

2.4 What Makes Parallel Programming Hard?

> Real difficulties can be overcome; it is only the imaginary ones that are unconquerable.
>
> *Theodore N. Vail*

It is important to note that the difficulty of parallel programming is as much a human-factors issue as it is a set of technical properties of the parallel programming problem. We do need human beings to be able to tell parallel systems what to do, otherwise known as programming. But parallel programming involves two-way communication, with a program's performance and scalability being the communication from the machine to the human. In short, the human writes a program telling the computer what to do, and the computer critiques this program via the resulting performance and scalability. Therefore, appeals to abstractions or to mathematical analyses will often be of severely limited utility.

In the Industrial Revolution, the interface between human and machine was evaluated by human-factor studies, then called time-and-motion studies. Although there have been a few human-factor studies examining parallel programming [ENS05, ES05, HCS+05, SS94], these studies have been extremely narrowly focused, and hence unable

www.dbooks.org

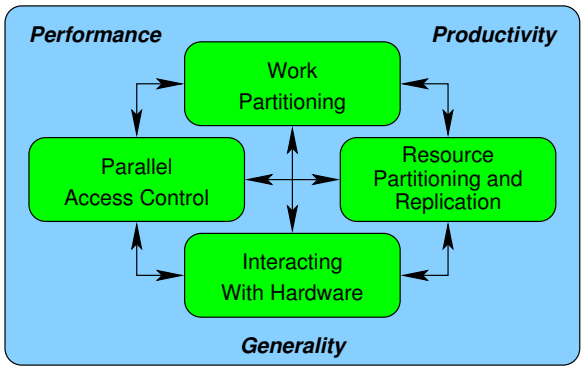


Figure 2.5: Categories of Tasks Required of Parallel Programmers

to demonstrate any general results. Furthermore, given that the normal range of programmer productivity spans more than an order of magnitude, it is unrealistic to expect an affordable study to be capable of detecting (say) a 10 % difference in productivity. Although the multiple-order-of-magnitude differences that such studies *can* reliably detect are extremely valuable, the most impressive improvements tend to be based on a long series of 10 % improvements.

We must therefore take a different approach.

One such approach is to carefully consider the tasks that parallel programmers must undertake that are not required of sequential programmers. We can then evaluate how well a given programming language or environment assists the developer with these tasks. These tasks fall into the four categories shown in Figure 2.5, each of which is covered in the following sections.

2.4.1 Work Partitioning

Work partitioning is absolutely required for parallel execution: if there is but one "glob" of work, then it can be executed by at most one CPU at a time, which is by definition sequential execution. However, partitioning the code requires great care. For example, uneven partitioning can result in sequential execution once the small partitions have completed [Amd67]. In less extreme cases, load balancing can be used to fully utilize available hardware and restore performance and scalabilty.

Although partitioning can greatly improve performance and scalability, it can also increase complexity. For example, partitioning can complicate handling of global errors and events: A parallel program may need to carry out non-trivial synchronization in order to safely process such global events. More generally, each partition requires some sort of communication: After all, if a given thread did not communicate at all, it would have no effect and would thus not need to be executed. However, because communication incurs overhead, careless partitioning choices can result in severe performance degradation.

Furthermore, the number of concurrent threads must often be controlled, as each such thread occupies common resources, for example, space in CPU caches. If too many threads are permitted to execute concurrently, the CPU caches will overflow, resulting in high cache miss rate, which in turn degrades performance. Conversely, large numbers of threads are often required to overlap computation and I/O so as to fully utilize I/O devices.

Quick Quiz 2.14: Other than CPU cache capacity, what might require limiting the number of concurrent threads? ■

Finally, permitting threads to execute concurrently greatly increases the program's state space, which can make the program difficult to understand and debug, degrading productivity. All else being equal, smaller state spaces having more regular structure are more easily understood, but this is a human-factors statement as much as it is a technical or mathematical statement. Good parallel designs might have extremely large state spaces, but nevertheless be easy to understand due to their regular structure, while poor designs can be impenetrable despite having a comparatively small state space. The best designs exploit embarrassing parallelism, or transform the problem to one having an embarrassingly parallel solution. In either case, "embarrassingly parallel" is in fact an embarrassment of riches. The current state of the art enumerates good designs; more work is required to make more general judgments on state-space size and structure.

2.4.2 Parallel Access Control

Given a single-threaded sequential program, that single thread has full access to all of the program's resources. These resources are most often in-memory data structures, but can be CPUs, memory (including caches), I/O devices, computational accelerators, files, and much else besides.

The first parallel-access-control issue is whether the form of access to a given resource depends on that resource's location. For example, in many message-passing environments, local-variable access is via expressions and assignments, while remote-variable access uses an entirely different syntax, usually involving messaging. The POSIX Threads environment [Ope97], Structured Query Language (SQL) [Int92], and partitioned global address-space (PGAS) environments such as Universal

Parallel C (UPC) [EGCD03, CBF13] offer implicit access, while Message Passing Interface (MPI) [MPI08] offers explicit access because access to remote data requires explicit messaging.

The other parallel-access-control issue is how threads coordinate access to the resources. This coordination is carried out by the very large number of synchronization mechanisms provided by various parallel languages and environments, including message passing, locking, transactions, reference counting, explicit timing, shared atomic variables, and data ownership. Many traditional parallel-programming concerns such as deadlock, livelock, and transaction rollback stem from this coordination. This framework can be elaborated to include comparisons of these synchronization mechanisms, for example locking vs. transactional memory [MMW07], but such elaboration is beyond the scope of this section. (See Sections 17.2 and 17.3 for more information on transactional memory.)

Quick Quiz 2.15: Just what is "explicit timing"??? ■

2.4.3 Resource Partitioning and Replication

The most effective parallel algorithms and systems exploit resource parallelism, so much so that it is usually wise to begin parallelization by partitioning your write-intensive resources and replicating frequently accessed read-mostly resources. The resource in question is most frequently data, which might be partitioned over computer systems, mass-storage devices, NUMA nodes, CPU cores (or dies or hardware threads), pages, cache lines, instances of synchronization primitives, or critical sections of code. For example, partitioning over locking primitives is termed "data locking" [BK85].

Resource partitioning is frequently application dependent. For example, numerical applications frequently partition matrices by row, column, or sub-matrix, while commercial applications frequently partition write-intensive data structures and replicate read-mostly data structures. Thus, a commercial application might assign the data for a given customer to a given few computers out of a large cluster. An application might statically partition data, or dynamically change the partitioning over time.

Resource partitioning is extremely effective, but it can be quite challenging for complex multilinked data structures.

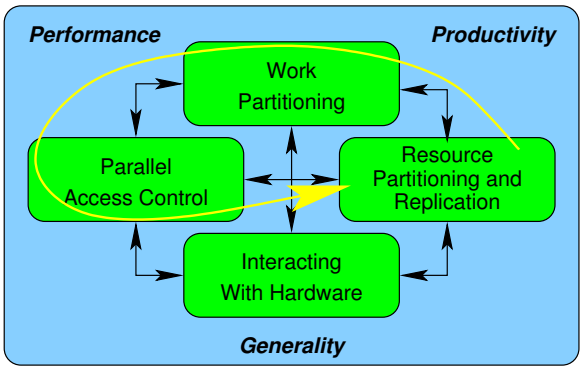


Figure 2.6: Ordering of Parallel-Programming Tasks

2.4.4 Interacting With Hardware

Hardware interaction is normally the domain of the operating system, the compiler, libraries, or other software-environment infrastructure. However, developers working with novel hardware features and components will often need to work directly with such hardware. In addition, direct access to the hardware can be required when squeezing the last drop of performance out of a given system. In this case, the developer may need to tailor or configure the application to the cache geometry, system topology, or interconnect protocol of the target hardware.

In some cases, hardware may be considered to be a resource which is subject to partitioning or access control, as described in the previous sections.

2.4.5 Composite Capabilities

Although these four capabilities are fundamental, good engineering practice uses composites of these capabilities. For example, the data-parallel approach first partitions the data so as to minimize the need for inter-partition communication, partitions the code accordingly, and finally maps data partitions and threads so as to maximize throughput while minimizing inter-thread communication, as shown in Figure 2.6. The developer can then consider each partition separately, greatly reducing the size of the relevant state space, in turn increasing productivity. Even though some problems are non-partitionable, clever transformations into forms permitting partitioning can sometimes greatly enhance both performance and scalability [Met99].

www.dbooks.org

2.4.6 How Do Languages and Environments Assist With These Tasks?

Although many environments require the developer to deal manually with these tasks, there are long-standing environments that bring significant automation to bear. The poster child for these environments is SQL, many implementations of which automatically parallelize single large queries and also automate concurrent execution of independent queries and updates.

These four categories of tasks must be carried out in all parallel programs, but that of course does not necessarily mean that the developer must manually carry out these tasks. We can expect to see ever-increasing automation of these four tasks as parallel systems continue to become cheaper and more readily available.

Quick Quiz 2.16: Are there any other obstacles to parallel programming? ■

2.5 Discussion

> Until you try, you don't know what you can't do.
>
> *Henry James*

This section has given an overview of the difficulties with, goals of, and alternatives to parallel programming. This overview was followed by a discussion of what can make parallel programming hard, along with a high-level approach for dealing with parallel programming's difficulties. Those who still insist that parallel programming is impossibly difficult should review some of the older guides to parallel programmming [Seq88, Bir89, BK85, Inm85]. The following quote from Andrew Birrell's monograph [Bir89] is especially telling:

> Writing concurrent programs has a reputation for being exotic and difficult. I believe it is neither. You need a system that provides you with good primitives and suitable libraries, you need a basic caution and carefulness, you need an armory of useful techniques, and you need to know of the common pitfalls. I hope that this paper has helped you towards sharing my belief.

The authors of these older guides were well up to the parallel programming challenge back in the 1980s. As such, there are simply no excuses for refusing to step up to the parallel-programming challenge here in the 21st century!

We are now ready to proceed to the next chapter, which dives into the relevant properties of the parallel hardware underlying our parallel software.

Premature abstraction is the root of all evil.

A cast of thousands

Chapter 3

Hardware and its Habits

Most people intuitively understand that passing messages between systems is more expensive than performing simple calculations within the confines of a single system. But it is also the case that communicating among threads within the confines of a single shared-memory system can also be quite expensive. This chapter therefore looks at the cost of synchronization and communication within a shared-memory system. These few pages can do no more than scratch the surface of shared-memory parallel hardware design; readers desiring more detail would do well to start with a recent edition of Hennessy's and Patterson's classic text [HP17, HP95].

Quick Quiz 3.1: Why should parallel programmers bother learning low-level properties of the hardware? Wouldn't it be easier, better, and more elegant to remain at a higher level of abstraction? ■

3.1 Overview

Mechanical Sympathy: Hardware and software working together in harmony.

Martin Thompson

Careless reading of computer-system specification sheets might lead one to believe that CPU performance is a footrace on a clear track, as illustrated in Figure 3.1, where the race always goes to the swiftest.

Although there are a few CPU-bound benchmarks that approach the ideal case shown in Figure 3.1, the typical program more closely resembles an obstacle course than a race track. This is because the internal architecture of CPUs has changed dramatically over the past few decades, courtesy of Moore's Law. These changes are described in the following sections.

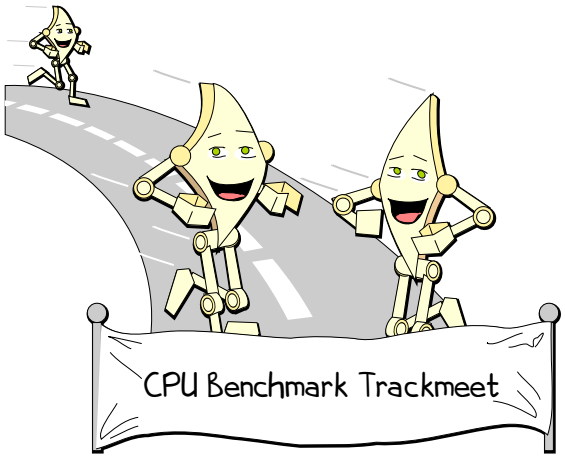


Figure 3.1: CPU Performance at its Best

3.1.1 Pipelined CPUs

In the 1980s, the typical microprocessor fetched an instruction, decoded it, and executed it, typically taking *at least* three clock cycles to complete one instruction before even starting the next. In contrast, the CPU of the late 1990s and of the 2000s execute many instructions simultaneously, using *pipelines*; *superscalar* techniques; *out-of-order* instruction and data handling; *speculative execution*, and more [HP17, HP11] in order to optimize the flow of instructions and data through the CPU. Some cores have more than one hardware thread, which is variously called *simultaneous multithreading* (SMT) or *hyperthreading* (HT) [Fen73], each of which appears as an independent CPU to software, at least from a functional viewpoint. These modern hardware features can greatly improve performance, as illustrated by Figure 3.2.

Achieving full performance with a CPU having a long pipeline requires highly predictable control flow through

www.dbooks.org

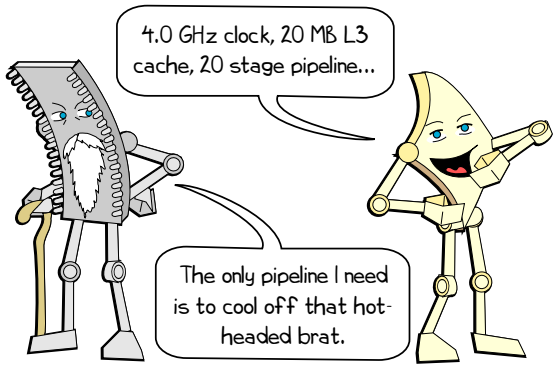


Figure 3.2: CPUs Old and New

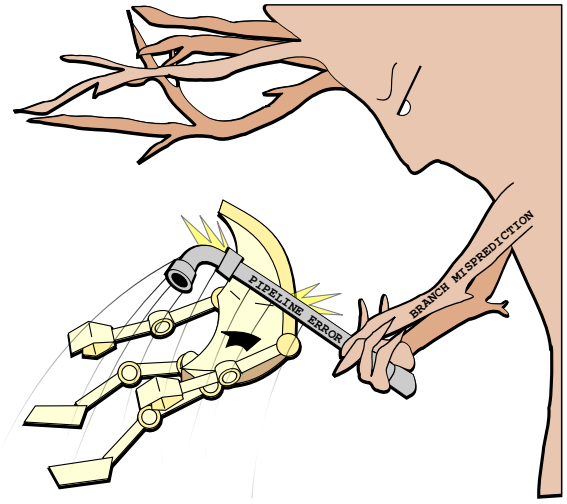


Figure 3.3: CPU Meets a Pipeline Flush

the program. Suitable control flow can be provided by a program that executes primarily in tight loops, for example, arithmetic on large matrices or vectors. The CPU can then correctly predict that the branch at the end of the loop will be taken in almost all cases, allowing the pipeline to be kept full and the CPU to execute at full speed.

However, branch prediction is not always so easy. For example, consider a program with many loops, each of which iterates a small but random number of times. For another example, consider an old-school object-oriented program with many virtual objects that can reference many different real objects, all with different implementations for frequently invoked member functions, resulting in many calls through pointers. In these cases, it is difficult or even impossible for the CPU to predict where the next branch might lead. Then either the CPU must stall waiting

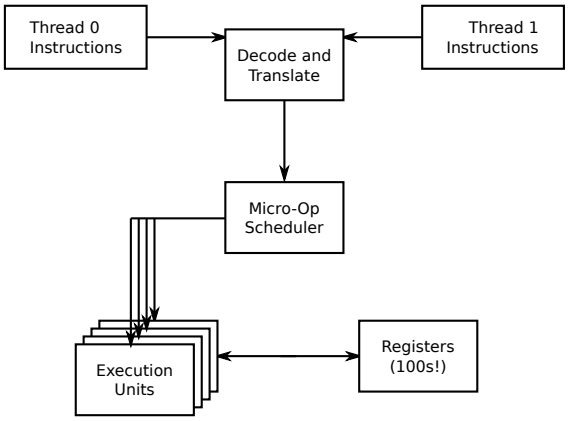


Figure 3.4: Rough View of Modern Micro-Architecture

for execution to proceed far enough to be certain where that branch leads, or it must guess and then proceed using speculative execution. Although guessing works extremely well for programs with predictable control flow, for unpredictable branches (such as those in binary search) the guesses will frequently be wrong. A wrong guess can be expensive because the CPU must discard any speculatively executed instructions following the corresponding branch, resulting in a pipeline flush. If pipeline flushes appear too frequently, they drastically reduce overall performance, as fancifully depicted in Figure 3.3.

This gets even worse in the increasingly common case of hyperthreading (or SMT, if you prefer), especially on pipelined superscalar out-of-order CPU featuring speculative execution. In this increasingly common case, all the hardware threads sharing a core also share that core's resources, including registers, cache, execution units, and so on. The instructions are often decoded into micro-operations, and use of the shared execution units and the hundreds of hardware registers is often coordinated by a micro-operation scheduler. A rough diagram of such a two-threaded core is shown in Figure 3.4, and more accurate (and thus more complex) diagrams are available in textbooks and scholarly papers.[1] Therefore, the execution of one hardware thread can and often is perturbed by the actions of other hardware threads sharing that core.

Even if only one hardware thread is active (for example, in old-school CPU designs where there is only one thread), counterintuitive results are quite common. Execution units often have overlapping capabilities, so that a CPU's

[1] Here is one example for a late-2010s Intel core: `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)`.

choice of execution unit can result in pipeline stalls due to contention for that execution unit from later instructions. In theory, this contention is avoidable, but in practice CPUs must choose very quickly and without the benefit of clairvoyance. In particular, adding an instruction to a tight loop can sometimes actually cause execution to *speed up*.

Unfortunately, pipeline flushes and shared-resource contention are not the only hazards in the obstacle course that modern CPUs must run. The next section covers the hazards of referencing memory.

3.1.2 Memory References

In the 1980s, it often took less time for a microprocessor to load a value from memory than it did to execute an instruction. More recently, microprocessors might execute hundreds or even thousands of instructions in the time required to access memory. This disparity is due to the fact that Moore's Law has increased CPU performance at a much greater rate than it has decreased memory latency, in part due to the rate at which memory sizes have grown. For example, a typical 1970s minicomputer might have 4 KB (yes, kilobytes, not megabytes, let alone gigabytes or terabytes) of main memory, with single-cycle access.[2] Present-day CPU designers still can construct a 4 KB memory with single-cycle access, even on systems with multi-GHz clock frequencies. And in fact they frequently do construct such memories, but they now call them "level-0 caches", plus they can be quite a bit bigger than 4 KB.

Although the large caches found on modern microprocessors can do quite a bit to help combat memory-access latencies, these caches require highly predictable data-access patterns to successfully hide those latencies. Unfortunately, common operations such as traversing a linked list have extremely unpredictable memory-access patterns—after all, if the pattern was predictable, us software types would not bother with the pointers, right? Therefore, as shown in Figure 3.5, memory references often pose severe obstacles to modern CPUs.

Thus far, we have only been considering obstacles that can arise during a given CPU's execution of single-threaded code. Multi-threading presents additional obstacles to the CPU, as described in the following sections.

[2] It is only fair to add that each of these single cycles lasted no less than 1.6 *microseconds*.

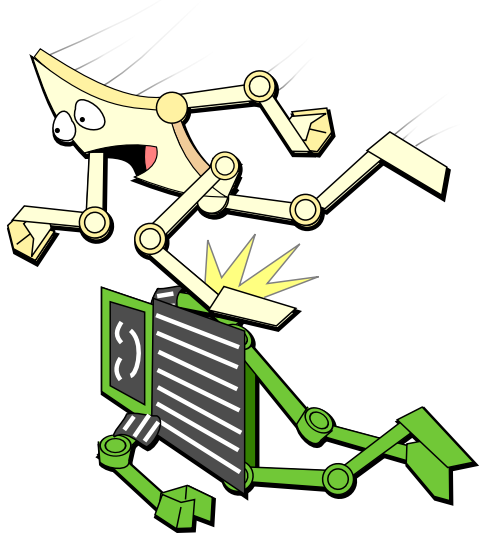


Figure 3.5: CPU Meets a Memory Reference

3.1.3 Atomic Operations

One such obstacle is atomic operations. The problem here is that the whole idea of an atomic operation conflicts with the piece-at-a-time assembly-line operation of a CPU pipeline. To hardware designers' credit, modern CPUs use a number of extremely clever tricks to make such operations *look* atomic even though they are in fact being executed piece-at-a-time, with one common trick being to identify all the cachelines containing the data to be atomically operated on, ensure that these cachelines are owned by the CPU executing the atomic operation, and only then proceed with the atomic operation while ensuring that these cachelines remained owned by this CPU. Because all the data is private to this CPU, other CPUs are unable to interfere with the atomic operation despite the piece-at-a-time nature of the CPU's pipeline. Needless to say, this sort of trick can require that the pipeline must be delayed or even flushed in order to perform the setup operations that permit a given atomic operation to complete correctly.

In contrast, when executing a non-atomic operation, the CPU can load values from cachelines as they appear and place the results in the store buffer, without the need to wait for cacheline ownership. Although there are a number of hardware optimizations that can sometimes hide cache latencies, the resulting effect on performance is all too often as depicted in Figure 3.6.

www.dbooks.org

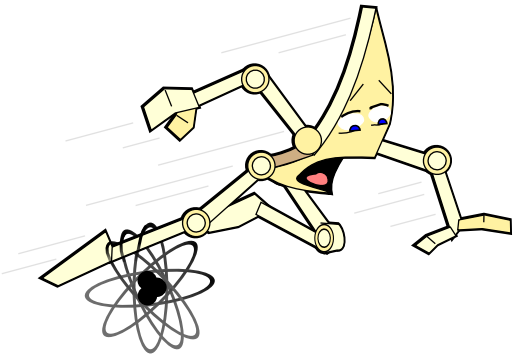


Figure 3.6: CPU Meets an Atomic Operation

Unfortunately, atomic operations usually apply only to single elements of data. Because many parallel algorithms require that ordering constraints be maintained between updates of multiple data elements, most CPUs provide memory barriers. These memory barriers also serve as performance-sapping obstacles, as described in the next section.

Quick Quiz 3.2: What types of machines would allow atomic operations on multiple data elements? ■

3.1.4 Memory Barriers

Memory barriers will be considered in more detail in Chapter 15 and Appendix C. In the meantime, consider the following simple lock-based critical section:

```
1 spin_lock(&mylock);
2 a = a + 1;
3 spin_unlock(&mylock);
```

If the CPU were not constrained to execute these statements in the order shown, the effect would be that the variable "a" would be incremented without the protection of "mylock", which would certainly defeat the purpose of acquiring it. To prevent such destructive reordering, locking primitives contain either explicit or implicit memory barriers. Because the whole purpose of these memory barriers is to prevent reorderings that the CPU would otherwise undertake in order to increase performance, memory barriers almost always reduce performance, as depicted in Figure 3.7.

As with atomic operations, CPU designers have been working hard to reduce memory-barrier overhead, and have made substantial progress.

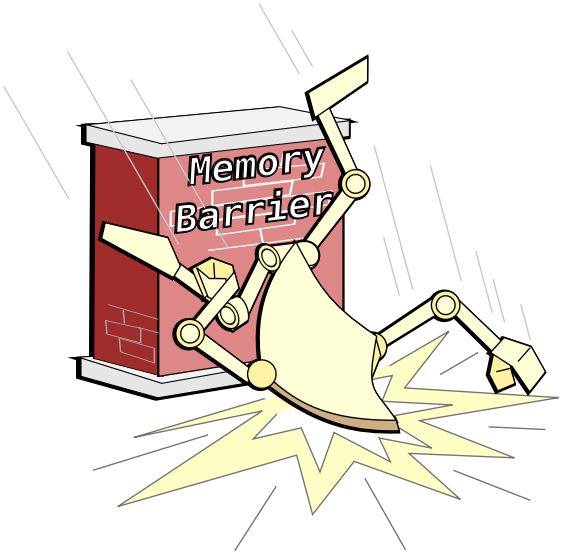


Figure 3.7: CPU Meets a Memory Barrier

3.1.5 Cache Misses

An additional multi-threading obstacle to CPU performance is the "cache miss". As noted earlier, modern CPUs sport large caches in order to reduce the performance penalty that would otherwise be incurred due to high memory latencies. However, these caches are actually counter-productive for variables that are frequently shared among CPUs. This is because when a given CPU wishes to modify the variable, it is most likely the case that some other CPU has modified it recently. In this case, the variable will be in that other CPU's cache, but not in this CPU's cache, which will therefore incur an expensive cache miss (see Section C.1 for more detail). Such cache misses form a major obstacle to CPU performance, as shown in Figure 3.8.

Quick Quiz 3.3: So have CPU designers also greatly reduced the overhead of cache misses? ■

3.1.6 I/O Operations

A cache miss can be thought of as a CPU-to-CPU I/O operation, and as such is one of the cheapest I/O operations available. I/O operations involving networking, mass storage, or (worse yet) human beings pose much greater obstacles than the internal obstacles called out in the prior sections, as illustrated by Figure 3.9.

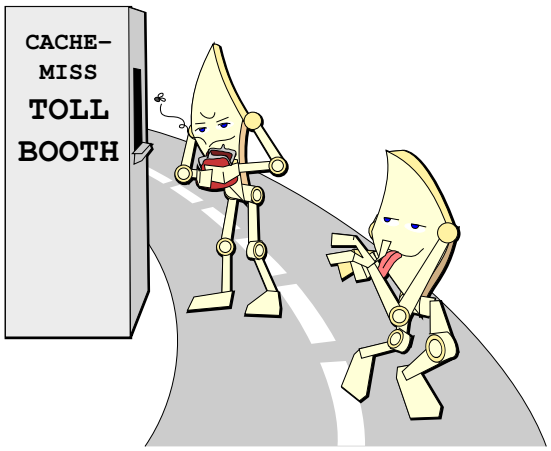


Figure 3.8: CPU Meets a Cache Miss

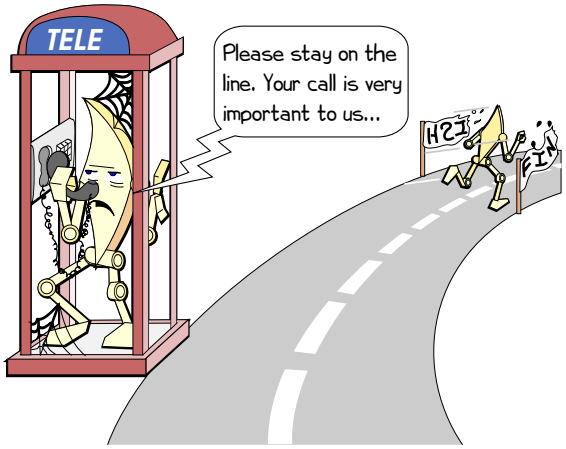


Figure 3.9: CPU Waits for I/O Completion

This is one of the differences between shared-memory and distributed-system parallelism: shared-memory parallel programs must normally deal with no obstacle worse than a cache miss, while a distributed parallel program will typically incur the larger network communication latencies. In both cases, the relevant latencies can be thought of as a cost of communication—a cost that would be absent in a sequential program. Therefore, the ratio between the overhead of the communication to that of the actual work being performed is a key design parameter. A major goal of parallel hardware design is to reduce this ratio as needed to achieve the relevant performance and scalability goals. In turn, as will be seen in Chapter 6, a major goal of parallel software design is to reduce the frequency of expensive operations like communications cache misses.

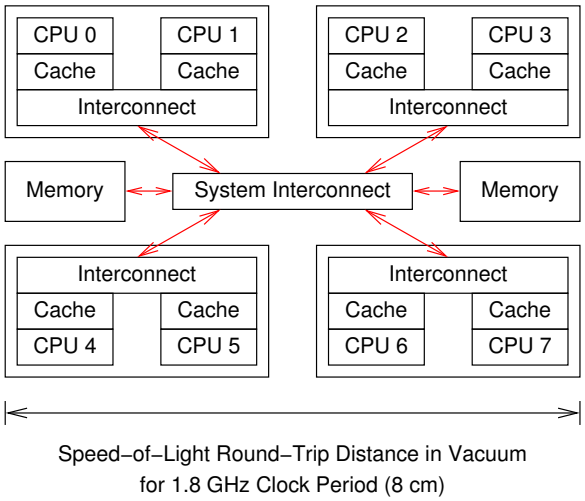


Figure 3.10: System Hardware Architecture

Of course, it is one thing to say that a given operation is an obstacle, and quite another to show that the operation is a *significant* obstacle. This distinction is discussed in the following sections.

3.2 Overheads

> Don't design bridges in ignorance of materials, and don't design low-level software in ignorance of the underlying hardware.
>
> *Unknown*

This section presents actual overheads of the obstacles to performance listed out in the previous section. However, it is first necessary to get a rough view of hardware system architecture, which is the subject of the next section.

3.2.1 Hardware System Architecture

Figure 3.10 shows a rough schematic of an eight-core computer system. Each die has a pair of CPU cores, each with its cache, as well as an interconnect allowing the pair of CPUs to communicate with each other. The system interconnect allows the four dies to communicate with each other and with main memory.

Data moves through this system in units of "cache lines", which are power-of-two fixed-size aligned blocks

www.dbooks.org

of memory, usually ranging from 32 to 256 bytes in size. When a CPU loads a variable from memory to one of its registers, it must first load the cacheline containing that variable into its cache. Similarly, when a CPU stores a value from one of its registers into memory, it must also load the cacheline containing that variable into its cache, but must also ensure that no other CPU has a copy of that cacheline.

For example, if CPU 0 were to write to a variable whose cacheline resided in CPU 7's cache, the following over-simplified sequence of events might ensue:

1. CPU 0 checks its local cache, and does not find the cacheline. It therefore records the write in its store buffer.
2. A request for this cacheline is forwarded to CPU 0's and 1's interconnect, which checks CPU 1's local cache, and does not find the cacheline.
3. This request is forwarded to the system interconnect, which checks with the other three dies, learning that the cacheline is held by the die containing CPU 6 and 7.
4. This request is forwarded to CPU 6's and 7's interconnect, which checks both CPUs' caches, finding the value in CPU 7's cache.
5. CPU 7 forwards the cacheline to its interconnect, and also flushes the cacheline from its cache.
6. CPU 6's and 7's interconnect forwards the cacheline to the system interconnect.
7. The system interconnect forwards the cacheline to CPU 0's and 1's interconnect.
8. CPU 0's and 1's interconnect forwards the cacheline to CPU 0's cache.
9. CPU 0 can now complete the write, updating the relevant portions of the newly arrived cacheline from the value previously recorded in the store buffer.

Quick Quiz 3.4: This is a *simplified* sequence of events? How could it *possibly* be any more complex? ■

Quick Quiz 3.5: Why is it necessary to flush the cacheline from CPU 7's cache? ■

This simplified sequence is just the beginning of a discipline called *cache-coherency protocols* [HP95, CSG99, MHS12, SHW11], which is discussed in more detail in Appendix C. As can be seen in the sequence of events triggered by a CAS operation, a single instruction can cause considerable protocol traffic, which can significantly degrade your parallel program's performance.

Fortunately, if a given variable is being frequently read during a time interval during which it is never updated, that variable can be replicated across all CPUs' caches. This replication permits all CPUs to enjoy extremely fast access to this *read-mostly* variable. Chapter 9 presents synchronization mechanisms that take full advantage of this important hardware read-mostly optimization.

3.2.2 Costs of Operations

The overheads of some common operations important to parallel programs are displayed in Table 3.1. This system's clock period rounds to 0.5 ns. Although it is not unusual for modern microprocessors to be able to retire multiple instructions per clock period, the operations' costs are nevertheless normalized to a clock period in the third column, labeled "Ratio". The first thing to note about this table is the large values of many of the ratios.

The same-CPU compare-and-swap (CAS) operation consumes about seven nanoseconds, a duration more than ten times that of the clock period. CAS is an atomic operation in which the hardware compares the contents of the specified memory location to a specified "old" value, and if they compare equal, stores a specified "new" value, in which case the CAS operation succeeds. If they compare unequal, the memory location keeps its (unexpected) value, and the CAS operation fails. The operation is atomic in that the hardware guarantees that the memory location will not be changed between the compare and the store. CAS functionality is provided by the `lock;cmpxchg` instruction on x86.

The "same-CPU" prefix means that the CPU now performing the CAS operation on a given variable was also the last CPU to access this variable, so that the corresponding cacheline is already held in that CPU's cache. Similarly, the same-CPU lock operation (a "round trip" pair consisting of a lock acquisition and release) consumes more than fifteen nanoseconds, or more than thirty clock cycles. The lock operation is more expensive than CAS because it requires two atomic operations on the lock data structure, one for acquisition and the other for release.

In-core operations involving interactions between the hardware threads sharing a single core are about the same cost as same-CPU operations. This should not be too

Table 3.1: CPU 0 View of Synchronization Mechanisms on 8-Socket System With Intel Xeon Platinum 8176 CPUs @ 2.10 GHz

Operation	Cost (ns)	Ratio (cost/clock)	CPUs
Clock period	0.5	1.0	
Same-CPU CAS	7.0	14.6	0
Same-CPU lock	15.4	32.3	0
In-core blind CAS	7.2	15.2	224
In-core CAS	18.0	37.7	224
Off-core blind CAS	47.5	99.8	1–27,225–251
Off-core CAS	101.9	214.0	1–27,225–251
Off-socket blind CAS	148.8	312.5	28–111,252–335
Off-socket CAS	442.9	930.1	28–111,252–335
Cross-interconnect blind CAS	336.6	706.8	112–223,336–447
Cross-interconnect CAS	944.8	1,984.2	112–223,336–447
Off-System			
Comms Fabric	5,000	10,500	
Global Comms	195,000,000	409,500,000	

surprising, given that these two hardware threads also share the full cache hierarchy.

In the case of the blind CAS, the software specifies the old value without looking at the memory location. This approach is appropriate when attempting to acquire a lock. If the unlocked state is represented by zero and the locked state is represented by the value one, then a CAS operation on the lock that specifies zero for the old value and one for the new value will acquire the lock if it is not already held. The key point is that there is only one access to the memory location, namely the CAS operation itself.

In contrast, a normal CAS operation's old value is derived from some earlier load. For example, to implement an atomic increment, the current value of that location is loaded and that value is incremented to produce the new value. Then in the CAS operation, the value actually loaded would be specified as the old value and the incremented value as the new value. If the value had not been changed between the load and the CAS, this would increment the memory location. However, if the value had in fact changed, then the old value would not match, causing a miscompare that would result in the CAS operation failing. The key point is that there are now two accesses to the memory location, the load and the CAS.

Thus, it is not surprising that in-core blind CAS consumes only about seven nanoseconds, while in-core CAS consumes about 18 nanoseconds. The non-blind case's extra load does not come for free. That said, the overhead of these operations are similar to single-CPU CAS and lock, respectively.

Quick Quiz 3.6: Table 3.1 shows CPU 0 sharing a core with CPU 224. Shouldn't that instead be CPU 1??? ■

A blind CAS involving CPUs in different cores but on the same socket consumes almost fifty nanoseconds, or almost one hundred clock cycles. The code used for this cache-miss measurement passes the cache line back and forth between a pair of CPUs, so this cache miss is satisfied not from memory, but rather from the other CPU's cache. A non-blind CAS operation, which as noted earlier must look at the old value of the variable as well as store a new value, consumes over one hundred nanoseconds, or more than two hundred clock cycles. Think about this a bit. In the time required to do *one* CAS operation, the CPU could have executed more than *two hundred* normal instructions. This should demonstrate the limitations not only of fine-grained locking, but of any other synchronization mechanism relying on fine-grained global agreement.

If the pair of CPUs are on different sockets, the operations are considerably more expensive. A blind CAS operation consumes almost 150 nanoseconds, or more than three hundred clock cycles. A normal CAS operation consumes more than 400 nanoseconds, or almost *one thousand* clock cycles.

www.dbooks.org

Table 3.2: Cache Geometry for 8-Socket System With Intel Xeon Platinum 8176 CPUs @ 2.10 GHz

Level	Scope	Line Size	Sets	Ways	Size
L0	Core	64	64	8	32K
L1	Core	64	64	8	32K
L2	Core	64	1024	16	1024K
L3	Socket	64	57,344	11	39,424K

Worse yet, not all pairs of sockets are created equal. This particular system appears to be constructed as a pair of four-socket components, with additional latency penalties when the CPUs reside in different components. In this case, a blind CAS operation consumes more than three hundred nanoseconds, or more than seven hundred clock cycles. A CAS operation consumes almost a full microsecond, or almost two thousand clock cycles.

Quick Quiz 3.7: Surely the hardware designers could be persuaded to improve this situation! Why have they been content with such abysmal performance for these single-instruction operations? ■

Unfortunately, the high speed of within-core and within-socket communication does not come for free. First, there are only two CPUs within a given core and only 56 within a given socket, compared to 448 across the system. Second, as shown in Table 3.2, the in-core caches are quite small compared to the in-socket caches, which are in turn quite small compared to the 1.4 TB of memory configured on this system. Third, again referring to the figure, the caches are organized as a hardware hash table with a limited number of items per bucket. For example, the raw size of the L3 cache ("Size") is almost 40 MB, but each bucket ("Line") can only hold 11 blocks of memory ("Ways"), each of which can be at most 64 bytes ("Line Size"). This means that only 12 bytes of memory (admittedly at carefully chosen addresses) are required to overflow this 40 MB cache. On the other hand, equally careful choice of addresses might make good use of the entire 40 MB.

Spatial locality of reference is clearly extremely important, as is spreading the data across memory.

I/O operations are even more expensive. As shown in the "Comms Fabric" row, high performance (and expensive!) communications fabric, such as InfiniBand or any number of proprietary interconnects, has a latency of roughly five microseconds for an end-to-end round trip, during which time more than *ten thousand* instructions might have been executed. Standards-based communications networks often require some sort of protocol processing, which further increases the latency. Of course, geographic distance also increases latency, with the speed-of-light through optical fiber latency around the world coming to roughly 195 *milliseconds*, or more than 400 million clock cycles, as shown in the "Global Comms" row.

Quick Quiz 3.8: These numbers are insanely large! How can I possibly get my head around them? ■

3.2.3 Hardware Optimizations

It is only natural to ask how the hardware is helping, and the answer is "Quite a bit!"

One hardware optimization is large cachelines. This can provide a big performance boost, especially when software is accessing memory sequentially. For example, given a 64-byte cacheline and software accessing 64-bit variables, the first access will still be slow due to speed-of-light delays (if nothing else), but the remaining seven can be quite fast. However, this optimization has a dark side, namely false sharing, which happens when different variables in the same cacheline are being updated by different CPUs, resulting in a high cache-miss rate. Software can use the alignment directives available in many compilers to avoid false sharing, and adding such directives is a common step in tuning parallel software.

A second related hardware optimization is cache prefetching, in which the hardware reacts to consecutive accesses by prefetching subsequent cachelines, thereby evading speed-of-light delays for these subsequent cachelines. Of course, the hardware must use simple heuristics to determine when to prefetch, and these heuristics can be fooled by the complex data-access patterns in many applications. Fortunately, some CPU families allow for this by providing special prefetch instructions. Unfortunately, the effectiveness of these instructions in the general case is subject to some dispute.

A third hardware optimization is the store buffer, which allows a string of store instructions to execute quickly even when the stores are to non-consecutive addresses and when none of the needed cachelines are present in the CPU's cache. The dark side of this optimization is memory misordering, for which see Chapter 15.

A fourth hardware optimization is speculative execution, which can allow the hardware to make good use of the store buffers without resulting in memory misordering. The dark side of this optimization can be energy inefficiency and lowered performance if the speculative execution goes awry and must be rolled back and retried. Worse yet, the

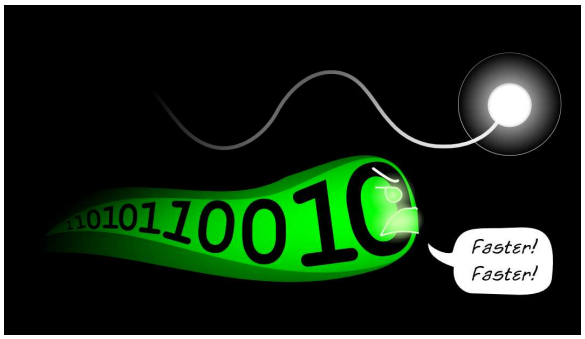


Figure 3.11: Hardware and Software: On Same Side

advent of Spectre and Meltdown [Hor18] made it apparent that hardware speculation can also enable side-channel attacks that defeat memory-protection hardware so as to allow unprivileged processes to read memory that they should not have access to. It is clear that the combination of speculative execution and cloud computing needs more than a bit of rework!

A fifth hardware optimization is large caches, allowing individual CPUs to operate on larger datasets without incurring expensive cache misses. Although large caches can degrade energy efficiency and cache-miss latency, the ever-growing cache sizes on production microprocessors attests to the power of this optimization.

A final hardware optimization is read-mostly replication, in which data that is frequently read but rarely updated is present in all CPUs' caches. This optimization allows the read-mostly data to be accessed exceedingly efficiently, and is the subject of Chapter 9.

In short, hardware and software engineers are really on the same side, with both trying to make computers go fast despite the best efforts of the laws of physics, as fancifully depicted in Figure 3.11 where our data stream is trying its best to exceed the speed of light. The next section discusses some additional things that the hardware engineers might (or might not) be able to do, depending on how well recent research translates to practice. Software's contribution to this noble goal is outlined in the remaining chapters of this book.

3.3 Hardware Free Lunch?

> The great trouble today is that there are too many people looking for someone else to do something for them. The solution to most of our troubles is to be found in everyone doing something for themselves.
>
> *Henry Ford, updated*

The major reason that concurrency has been receiving so much focus over the past few years is the end of Moore's-Law induced single-threaded performance increases (or "free lunch" [Sut08]), as shown in Figure 2.1 on page 9. This section briefly surveys a few ways that hardware designers might bring back the "free lunch".

However, the preceding section presented some substantial hardware obstacles to exploiting concurrency. One severe physical limitation that hardware designers face is the finite speed of light. As noted in Figure 3.10 on page 21, light can manage only about an 8-centimeters round trip in a vacuum during the duration of a 1.8 GHz clock period. This distance drops to about 3 centimeters for a 5 GHz clock. Both of these distances are relatively small compared to the size of a modern computer system.

To make matters even worse, electric waves in silicon move from three to thirty times more slowly than does light in a vacuum, and common clocked logic constructs run still more slowly, for example, a memory reference may need to wait for a local cache lookup to complete before the request may be passed on to the rest of the system. Furthermore, relatively low speed and high power drivers are required to move electrical signals from one silicon die to another, for example, to communicate between a CPU and main memory.

Quick Quiz 3.9: But individual electrons don't move anywhere near that fast, even in conductors!!! The electron drift velocity in a conductor under semiconductor voltage levels is on the order of only one *millimeter* per second. What gives???
■

There are nevertheless some technologies (both hardware and software) that might help improve matters:

1. 3D integration,
2. Novel materials and processes,
3. Substituting light for electricity,
4. Special-purpose accelerators, and
5. Existing parallel software.

www.dbooks.org

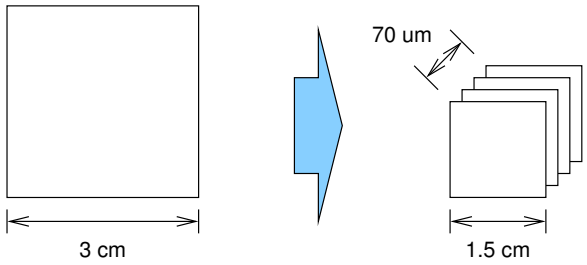


Figure 3.12: Latency Benefit of 3D Integration

Each of these is described in one of the following sections.

3.3.1 3D Integration

3-dimensional integration (3DI) is the practice of bonding very thin silicon dies to each other in a vertical stack. This practice provides potential benefits, but also poses significant fabrication challenges [Kni08].

Perhaps the most important benefit of 3DI is decreased path length through the system, as shown in Figure 3.12. A 3-centimeter silicon die is replaced with a stack of four 1.5-centimeter dies, in theory decreasing the maximum path through the system by a factor of two, keeping in mind that each layer is quite thin. In addition, given proper attention to design and placement, long horizontal electrical connections (which are both slow and power hungry) can be replaced by short vertical electrical connections, which are both faster and more power efficient.

However, delays due to levels of clocked logic will not be decreased by 3D integration, and significant manufacturing, testing, power-supply, and heat-dissipation problems must be solved for 3D integration to reach production while still delivering on its promise. The heat-dissipation problems might be solved using semiconductors based on diamond, which is a good conductor for heat, but an electrical insulator. That said, it remains difficult to grow large single diamond crystals, to say nothing of slicing them into wafers. In addition, it seems unlikely that any of these technologies will be able to deliver the exponential increases to which some people have become accustomed. That said, they may be necessary steps on the path to the late Jim Gray's "smoking hairy golf balls" [Gra02].

3.3.2 Novel Materials and Processes

Stephen Hawking is said to have claimed that semiconductor manufacturers have but two fundamental problems: (1) the finite speed of light and (2) the atomic nature of matter [Gar07]. It is possible that semiconductor manufacturers are approaching these limits, but there are nevertheless a few avenues of research and development focused on working around these fundamental limits.

One workaround for the atomic nature of matter are so-called "high-K dielectric" materials, which allow larger devices to mimic the electrical properties of infeasibly small devices. These materials pose some severe fabrication challenges, but nevertheless may help push the frontiers out a bit farther. Another more-exotic workaround stores multiple bits in a single electron, relying on the fact that a given electron can exist at a number of energy levels. It remains to be seen if this particular approach can be made to work reliably in production semiconductor devices.

Another proposed workaround is the "quantum dot" approach that allows much smaller device sizes, but which is still in the research stage.

One challenge is that many recent hardware-device-level breakthroughs require very tight control of which atoms are placed where [Kel17]. It therefore seems likely that whoever finds a good way to hand-place atoms on each of the billions of devices on a chip will have most excellent bragging rights, if nothing else!

3.3.3 Light, Not Electrons

Although the speed of light would be a hard limit, the fact is that semiconductor devices are limited by the speed of electricity rather than that of light, given that electric waves in semiconductor materials move at between 3 % and 30 % of the speed of light in a vacuum. The use of copper connections on silicon devices is one way to increase the speed of electricity, and it is quite possible that additional advances will push closer still to the actual speed of light. In addition, there have been some experiments with tiny optical fibers as interconnects within and between chips, based on the fact that the speed of light in glass is more than 60 % of the speed of light in a vacuum. One obstacle to such optical fibers is the inefficiency conversion between electricity and light and vice versa, resulting in both power-consumption and heat-dissipation problems.

That said, absent some fundamental advances in the field of physics, any exponential increases in the speed of data flow will be sharply limited by the actual speed of light in a vacuum.

3.3.4 Special-Purpose Accelerators

A general-purpose CPU working on a specialized problem is often spending significant time and energy doing work that is only tangentially related to the problem at hand. For example, when taking the dot product of a pair of vectors, a general-purpose CPU will normally use a loop (possibly unrolled) with a loop counter. Decoding the instructions, incrementing the loop counter, testing this counter, and branching back to the top of the loop are in some sense wasted effort: the real goal is instead to multiply corresponding elements of the two vectors. Therefore, a specialized piece of hardware designed specifically to multiply vectors could get the job done more quickly and with less energy consumed.

This is in fact the motivation for the vector instructions present in many commodity microprocessors. Because these instructions operate on multiple data items simultaneously, they would permit a dot product to be computed with less instruction-decode and loop overhead.

Similarly, specialized hardware can more efficiently encrypt and decrypt, compress and decompress, encode and decode, and many other tasks besides. Unfortunately, this efficiency does not come for free. A computer system incorporating this specialized hardware will contain more transistors, which will consume some power even when not in use. Software must be modified to take advantage of this specialized hardware, and this specialized hardware must be sufficiently generally useful that the high up-front hardware-design costs can be spread over enough users to make the specialized hardware affordable. In part due to these sorts of economic considerations, specialized hardware has thus far appeared only for a few application areas, including graphics processing (GPUs), vector processors (MMX, SSE, and VMX instructions), and, to a lesser extent, encryption. And even in these areas, it is not always easy to realize the expected performance gains, for example, due to thermal throttling [Kra17, Lem18, Dow20].

Unlike the server and PC arena, smartphones have long used a wide variety of hardware accelerators. These hardware accelerators are often used for media decoding, so much so that a high-end MP3 player might be able to play audio for several minutes—with its CPU fully powered off the entire time. The purpose of these accelerators is to improve energy efficiency and thus extend battery life: special purpose hardware can often compute more efficiently than can a general-purpose CPU. This is another example of the principle called out in Section 2.2.3: Generality is almost never free.

Nevertheless, given the end of Moore's-Law-induced single-threaded performance increases, it seems safe to assume that increasing varieties of special-purpose hardware will appear.

3.3.5 Existing Parallel Software

Although multicore CPUs seem to have taken the computing industry by surprise, the fact remains that shared-memory parallel computer systems have been commercially available for more than a quarter century. This is more than enough time for significant parallel software to make its appearance, and it indeed has. Parallel operating systems are quite commonplace, as are parallel threading libraries, parallel relational database management systems, and parallel numerical software. Use of existing parallel software can go a long ways towards solving any parallel-software crisis we might encounter.

Perhaps the most common example is the parallel relational database management system. It is not unusual for single-threaded programs, often written in high-level scripting languages, to access a central relational database concurrently. In the resulting highly parallel system, only the database need actually deal directly with parallelism. A very nice trick when it works!

3.4 Software Design Implications

> One ship drives east and another west
> While the self-same breezes blow;
> 'Tis the set of the sail and not the gail
> That bids them where to go.
>
> *Ella Wheeler Wilcox*

The values of the ratios in Table 3.1 are critically important, as they limit the efficiency of a given parallel application. To see this, suppose that the parallel application uses CAS operations to communicate among threads. These CAS operations will typically involve a cache miss, that is, assuming that the threads are communicating primarily with each other rather than with themselves. Suppose further that the unit of work corresponding to each CAS communication operation takes 300 ns, which is sufficient time to compute several floating-point transcendental functions. Then about half of the execution time will be consumed by the CAS communication operations! This in turn means that a two-CPU system running such a

www.dbooks.org

parallel program would run no faster than a sequential implementation running on a single CPU.

The situation is even worse in the distributed-system case, where the latency of a single communications operation might take as long as thousands or even millions of floating-point operations. This illustrates how important it is for communications operations to be extremely infrequent and to enable very large quantities of processing.

Quick Quiz 3.10: Given that distributed-systems communication is so horribly expensive, why does anyone bother with such systems? ■

The lesson should be quite clear: parallel algorithms must be explicitly designed with these hardware properties firmly in mind. One approach is to run nearly independent threads. The less frequently the threads communicate, whether by atomic operations, locks, or explicit messages, the better the application's performance and scalability will be. This approach will be touched on in Chapter 5, explored in Chapter 6, and taken to its logical extreme in Chapter 8.

Another approach is to make sure that any sharing be read-mostly, which allows the CPUs' caches to replicate the read-mostly data, in turn allowing all CPUs fast access. This approach is touched on in Section 5.2.4, and explored more deeply in Chapter 9.

In short, achieving excellent parallel performance and scalability means striving for embarrassingly parallel algorithms and implementations, whether by careful choice of data structures and algorithms, use of existing parallel applications and environments, or transforming the problem into an embarrassingly parallel form.

Quick Quiz 3.11: OK, if we are going to have to apply distributed-programming techniques to shared-memory parallel programs, why not just always use these distributed techniques and dispense with shared memory? ■

So, to sum up:

1. The good news is that multicore systems are inexpensive and readily available.
2. More good news: The overhead of many synchronization operations is much lower than it was on parallel systems from the early 2000s.
3. The bad news is that the overhead of cache misses is still high, especially on large systems.

The remainder of this book describes ways of handling this bad news.

In particular, Chapter 4 will cover some of the low-level tools used for parallel programming, Chapter 5 will investigate problems and solutions to parallel counting, and Chapter 6 will discuss design disciplines that promote performance and scalability.

You are only as good as your tools, and your tools are only as good as you are.

Unknown

Chapter 4

Tools of the Trade

This chapter provides a brief introduction to some basic tools of the parallel-programming trade, focusing mainly on those available to user applications running on operating systems similar to Linux. Section 4.1 begins with scripting languages, Section 4.2 describes the multi-process parallelism supported by the POSIX API and touches on POSIX threads, Section 4.3 presents analogous operations in other environments, and finally, Section 4.4 helps to choose the tool that will get the job done.

Quick Quiz 4.1: You call these tools??? They look more like low-level synchronization primitives to me! ■

Please note that this chapter provides but a brief introduction. More detail is available from the references (and from the Internet), and more information will be provided in later chapters.

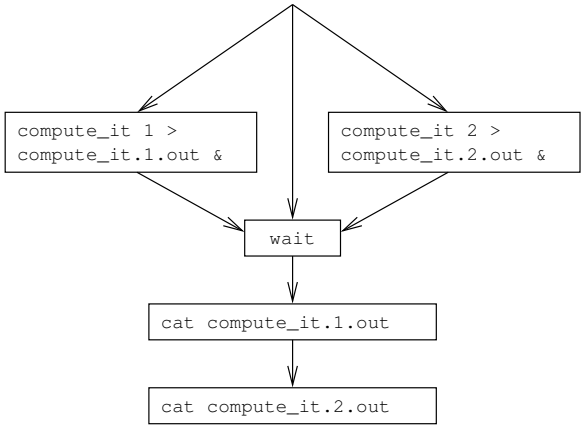


Figure 4.1: Execution Diagram for Parallel Shell Execution

4.1 Scripting Languages

The supreme excellence is simplicity.

Henry Wadsworth Longfellow, abbreviated

The Linux shell scripting languages provide simple but effective ways of managing parallelism. For example, suppose that you had a program `compute_it` that you needed to run twice with two different sets of arguments. This can be accomplished using UNIX shell scripting as follows:

```
1 compute_it 1 > compute_it.1.out &
2 compute_it 2 > compute_it.2.out &
3 wait
4 cat compute_it.1.out
5 cat compute_it.2.out
```

Lines 1 and 2 launch two instances of this program, redirecting their output to two separate files, with the `&` character directing the shell to run the two instances of the program in the background. Line 3 waits for both instances to complete, and lines 4 and 5 display their output. The resulting execution is as shown in Figure 4.1: the two instances of `compute_it` execute in parallel, `wait` completes after both of them do, and then the two instances of `cat` execute sequentially.

Quick Quiz 4.2: But this silly shell script isn't a *real* parallel program! Why bother with such trivia??? ■

Quick Quiz 4.3: Is there a simpler way to create a parallel shell script? If so, how? If not, why not? ■

For another example, the `make` software-build scripting language provides a `-j` option that specifies how much parallelism should be introduced into the build process. Thus, typing `make -j4` when building a Linux kernel specifies that up to four build steps be executed concurrently.

www.dbooks.org

It is hoped that these simple examples convince you that parallel programming need not always be complex or difficult.

Quick Quiz 4.4: But if script-based parallel programming is so easy, why bother with anything else? ■

4.2 POSIX Multiprocessing

> A camel is a horse designed by committee.
>
> *Unknown*

This section scratches the surface of the POSIX environment, including pthreads [Ope97], as this environment is readily available and widely implemented. Section 4.2.1 provides a glimpse of the POSIX `fork()` and related primitives, Section 4.2.2 touches on thread creation and destruction, Section 4.2.3 gives a brief overview of POSIX locking, and, finally, Section 4.2.4 describes a specific lock which can be used for data that is read by many threads and only occasionally updated.

4.2.1 POSIX Process Creation and Destruction

Processes are created using the `fork()` primitive, they may be destroyed using the `kill()` primitive, they may destroy themselves using the `exit()` primitive. A process executing a `fork()` primitive is said to be the "parent" of the newly created process. A parent may wait on its children using the `wait()` primitive.

Please note that the examples in this section are quite simple. Real-world applications using these primitives might need to manipulate signals, file descriptors, shared memory segments, and any number of other resources. In addition, some applications need to take specific actions if a given child terminates, and might also need to be concerned with the reason that the child terminated. These issues can of course add substantial complexity to the code. For more information, see any of a number of textbooks on the subject [Ste92, Wei13].

If `fork()` succeeds, it returns twice, once for the parent and again for the child. The value returned from `fork()` allows the caller to tell the difference, as shown in Listing 4.1 (`forkjoin.c`). Line 1 executes the `fork()` primitive, and saves its return value in local variable `pid`. Line 2 checks to see if `pid` is zero, in which case, this is the child, which continues on to execute line 3. As noted earlier, the child may terminate via the `exit()` primitive. Otherwise, this is the parent, which checks for an error return from the `fork()` primitive on line 4, and prints an error and exits on lines 5–7 if so. Otherwise, the `fork()` has executed successfully, and the parent therefore executes line 9 with the variable `pid` containing the process ID of the child.

Listing 4.1: Using the `fork()` Primitive

```
1 pid = fork();
2 if (pid == 0) {
3   /* child */
4 } else if (pid < 0) {
5   /* parent, upon error */
6   perror("fork");
7   exit(EXIT_FAILURE);
8 } else {
9   /* parent, pid == child ID */
10 }
```

Listing 4.2: Using the `wait()` Primitive

```
1 static __inline__ void waitall(void)
2 {
3   int pid;
4   int status;
5
6   for (;;) {
7     pid = wait(&status);
8     if (pid == -1) {
9       if (errno == ECHILD)
10        break;
11      perror("wait");
12      exit(EXIT_FAILURE);
13    }
14  }
15 }
```

The parent process may use the `wait()` primitive to wait for its children to complete. However, use of this primitive is a bit more complicated than its shell-script counterpart, as each invocation of `wait()` waits for but one child process. It is therefore customary to wrap `wait()` into a function similar to the `waitall()` function shown in Listing 4.2 (`api-pthreads.h`), with this `waitall()` function having semantics similar to the shell-script `wait` command. Each pass through the loop spanning lines 6–14 waits on one child process. Line 7 invokes the `wait()` primitive, which blocks until a child process exits, and returns that child's process ID. If the process ID is instead −1, this indicates that the `wait()` primitive was unable to wait on a child. If so, line 9 checks for the `ECHILD` errno, which indicates that there are no more child processes, so that line 10 exits the loop. Otherwise, lines 11 and 12 print an error and exit.

Quick Quiz 4.5: Why does this `wait()` primitive need to be so complicated? Why not just make it work like the shell-script `wait` does? ■

Listing 4.3: Processes Created Via `fork()` Do Not Share Memory

```
1  int x = 0;
2
3  int main(int argc, char *argv[])
4  {
5    int pid;
6
7    pid = fork();
8    if (pid == 0) { /* child */
9      x = 1;
10     printf("Child process set x=1\n");
11     exit(EXIT_SUCCESS);
12   }
13   if (pid < 0) { /* parent, upon error */
14     perror("fork");
15     exit(EXIT_FAILURE);
16   }
17
18   /* parent */
19
20   waitall();
21   printf("Parent process sees x=%d\n", x);
22
23   return EXIT_SUCCESS;
24 }
```

It is critically important to note that the parent and child do *not* share memory. This is illustrated by the program shown in Listing 4.3 (`forkjoinvar.c`), in which the child sets a global variable `x` to 1 on line 9, prints a message on line 10, and exits on line 11. The parent continues at line 20, where it waits on the child, and on line 21 finds that its copy of the variable `x` is still zero. The output is thus as follows:

```
Child process set x=1
Parent process sees x=0
```

Quick Quiz 4.6: Isn't there a lot more to `fork()` and `wait()` than discussed here? ■

The finest-grained parallelism requires shared memory, and this is covered in Section 4.2.2. That said, shared-memory parallelism can be significantly more complex than fork-join parallelism.

4.2.2 POSIX Thread Creation and Destruction

To create a thread within an existing process, invoke the `pthread_create()` primitive, for example, as shown on lines 16 and 17 of Listing 4.4 (`pcreate.c`). The first argument is a pointer to a `pthread_t` in which to store the ID of the thread to be created, the second `NULL` argument is a pointer to an optional `pthread_attr_t`, the third argument is the function (in this case, `mythread()`)

Listing 4.4: Threads Created Via `pthread_create()` Share Memory

```
1  int x = 0;
2
3  void *mythread(void *arg)
4  {
5    x = 1;
6    printf("Child process set x=1\n");
7    return NULL;
8  }
9
10 int main(int argc, char *argv[])
11 {
12   int en;
13   pthread_t tid;
14   void *vp;
15
16   if ((en = pthread_create(&tid, NULL,
17                            mythread, NULL)) != 0) {
18     fprintf(stderr, "pthread_create: %s\n", strerror(en));
19     exit(EXIT_FAILURE);
20   }
21
22   /* parent */
23
24   if ((en = pthread_join(tid, &vp)) != 0) {
25     fprintf(stderr, "pthread_join: %s\n", strerror(en));
26     exit(EXIT_FAILURE);
27   }
28   printf("Parent process sees x=%d\n", x);
29
30   return EXIT_SUCCESS;
31 }
```

that is to be invoked by the new thread, and the last `NULL` argument is the argument that will be passed to `mythread`.

In this example, `mythread()` simply returns, but it could instead call `pthread_exit()`.

Quick Quiz 4.7: If the `mythread()` function in Listing 4.4 can simply return, why bother with `pthread_exit()`? ■

The `pthread_join()` primitive, shown on line 24, is analogous to the fork-join `wait()` primitive. It blocks until the thread specified by the `tid` variable completes execution, either by invoking `pthread_exit()` or by returning from the thread's top-level function. The thread's exit value will be stored through the pointer passed as the second argument to `pthread_join()`. The thread's exit value is either the value passed to `pthread_exit()` or the value returned by the thread's top-level function, depending on how the thread in question exits.

The program shown in Listing 4.4 produces output as follows, demonstrating that memory is in fact shared between the two threads:

```
Child process set x=1
Parent process sees x=1
```

Note that this program carefully makes sure that only one of the threads stores a value to variable `x` at a time.

www.dbooks.org

Any situation in which one thread might be storing a value to a given variable while some other thread either loads from or stores to that same variable is termed a "data race". Because the C language makes no guarantee that the results of a data race will be in any way reasonable, we need some way of safely accessing and modifying data concurrently, such as the locking primitives discussed in the following section.

Quick Quiz 4.8: If the C language makes no guarantees in presence of a data race, then why does the Linux kernel have so many data races? Are you trying to tell me that the Linux kernel is completely broken??? ■

4.2.3 POSIX Locking

The POSIX standard allows the programmer to avoid data races via "POSIX locking". POSIX locking features a number of primitives, the most fundamental of which are `pthread_mutex_lock()` and `pthread_mutex_unlock()`. These primitives operate on locks, which are of type `pthread_mutex_t`. These locks may be declared statically and initialized with `PTHREAD_MUTEX_INITIALIZER`, or they may be allocated dynamically and initialized using the `pthread_mutex_init()` primitive. The demonstration code in this section will take the former course.

The `pthread_mutex_lock()` primitive "acquires" the specified lock, and the `pthread_mutex_unlock()` "releases" the specified lock. Because these are "exclusive" locking primitives, only one thread at a time may "hold" a given lock at a given time. For example, if a pair of threads attempt to acquire the same lock concurrently, one of the pair will be "granted" the lock first, and the other will wait until the first thread releases the lock. A simple and reasonably useful programming model permits a given data item to be accessed only while holding the corresponding lock [Hoa74].

Quick Quiz 4.9: What if I want several threads to hold the same lock at the same time? ■

This exclusive-locking property is demonstrated using the code shown in Listing 4.5 (`lock.c`). Line 1 defines and initializes a POSIX lock named `lock_a`, while line 2 similarly defines and initializes a lock named `lock_b`. Line 4 defines and initializes a shared variable `x`.

Lines 6–33 define a function `lock_reader()` which repeatedly reads the shared variable `x` while holding the lock specified by `arg`. Line 12 casts `arg` to a pointer to a

Listing 4.5: Demonstration of Exclusive Locks

```
1  pthread_mutex_t lock_a = PTHREAD_MUTEX_INITIALIZER;
2  pthread_mutex_t lock_b = PTHREAD_MUTEX_INITIALIZER;
3
4  int x = 0;
5
6  void *lock_reader(void *arg)
7  {
8    int en;
9    int i;
10   int newx = -1;
11   int oldx = -1;
12   pthread_mutex_t *pmlp = (pthread_mutex_t *)arg;
13
14   if ((en = pthread_mutex_lock(pmlp)) != 0) {
15     fprintf(stderr, "lock_reader:pthread_mutex_lock: %s\n",
16       strerror(en));
17     exit(EXIT_FAILURE);
18   }
19   for (i = 0; i < 100; i++) {
20     newx = READ_ONCE(x);
21     if (newx != oldx) {
22       printf("lock_reader(): x = %d\n", newx);
23     }
24     oldx = newx;
25     poll(NULL, 0, 1);
26   }
27   if ((en = pthread_mutex_unlock(pmlp)) != 0) {
28     fprintf(stderr, "lock_reader:pthread_mutex_unlock: %s\n",
29       strerror(en));
30     exit(EXIT_FAILURE);
31   }
32   return NULL;
33 }
34
35 void *lock_writer(void *arg)
36 {
37   int en;
38   int i;
39   pthread_mutex_t *pmlp = (pthread_mutex_t *)arg;
40
41   if ((en = pthread_mutex_lock(pmlp)) != 0) {
42     fprintf(stderr, "lock_writer:pthread_mutex_lock: %s\n",
43       strerror(en));
44     exit(EXIT_FAILURE);
45   }
46   for (i = 0; i < 3; i++) {
47     WRITE_ONCE(x, READ_ONCE(x) + 1);
48     poll(NULL, 0, 5);
49   }
50   if ((en = pthread_mutex_unlock(pmlp)) != 0) {
51     fprintf(stderr, "lock_writer:pthread_mutex_unlock: %s\n",
52       strerror(en));
53     exit(EXIT_FAILURE);
54   }
55   return NULL;
56 }
```

Listing 4.6: Demonstration of Same Exclusive Lock

```
1  printf("Creating two threads using same lock:\n");
2  en = pthread_create(&tid1, NULL, lock_reader, &lock_a);
3  if (en != 0) {
4    fprintf(stderr, "pthread_create: %s\n", strerror(en));
5    exit(EXIT_FAILURE);
6  }
7  en = pthread_create(&tid2, NULL, lock_writer, &lock_a);
8  if (en != 0) {
9    fprintf(stderr, "pthread_create: %s\n", strerror(en));
10   exit(EXIT_FAILURE);
11 }
12 if ((en = pthread_join(tid1, &vp)) != 0) {
13   fprintf(stderr, "pthread_join: %s\n", strerror(en));
14   exit(EXIT_FAILURE);
15 }
16 if ((en = pthread_join(tid2, &vp)) != 0) {
17   fprintf(stderr, "pthread_join: %s\n", strerror(en));
18   exit(EXIT_FAILURE);
19 }
```

Listing 4.7: Demonstration of Different Exclusive Locks

```
1  printf("Creating two threads w/different locks:\n");
2  x = 0;
3  en = pthread_create(&tid1, NULL, lock_reader, &lock_a);
4  if (en != 0) {
5    fprintf(stderr, "pthread_create: %s\n", strerror(en));
6    exit(EXIT_FAILURE);
7  }
8  en = pthread_create(&tid2, NULL, lock_writer, &lock_b);
9  if (en != 0) {
10   fprintf(stderr, "pthread_create: %s\n", strerror(en));
11   exit(EXIT_FAILURE);
12 }
13 if ((en = pthread_join(tid1, &vp)) != 0) {
14   fprintf(stderr, "pthread_join: %s\n", strerror(en));
15   exit(EXIT_FAILURE);
16 }
17 if ((en = pthread_join(tid2, &vp)) != 0) {
18   fprintf(stderr, "pthread_join: %s\n", strerror(en));
19   exit(EXIT_FAILURE);
20 }
```

`pthread_mutex_t`, as required by the `pthread_mutex_lock()` and `pthread_mutex_unlock()` primitives.

Quick Quiz 4.10: Why not simply make the argument to `lock_reader()` on line 6 of Listing 4.5 be a pointer to a `pthread_mutex_t`? ■

Quick Quiz 4.11: What is the `READ_ONCE()` on lines 20 and 47 and the `WRITE_ONCE()` on line 47 of Listing 4.5? ■

Lines 14–18 acquire the specified `pthread_mutex_t`, checking for errors and exiting the program if any occur. Lines 19–26 repeatedly check the value of `x`, printing the new value each time that it changes. Line 25 sleeps for one millisecond, which allows this demonstration to run nicely on a uniprocessor machine. Lines 27–31 release the `pthread_mutex_t`, again checking for errors and exiting the program if any occur. Finally, line 32 returns `NULL`, again to match the function type required by `pthread_create()`.

Quick Quiz 4.12: Writing four lines of code for each acquisition and release of a `pthread_mutex_t` sure seems painful! Isn't there a better way? ■

Lines 35–56 of Listing 4.5 show `lock_writer()`, which periodically updates the shared variable `x` while holding the specified `pthread_mutex_t`. As with `lock_reader()`, line 39 casts `arg` to a pointer to `pthread_mutex_t`, lines 41–45 acquire the specified lock, and lines 50–54 release it. While holding the lock, lines 46–49 increment the shared variable `x`, sleeping for five milliseconds between each increment. Finally, lines 50–54 release the lock.

Listing 4.6 shows a code fragment that runs `lock_reader()` and `lock_writer()` as threads using the same lock, namely, `lock_a`. Lines 2–6 create a thread running `lock_reader()`, and then lines 7–11 create a thread running `lock_writer()`. Lines 12–19 wait for both threads to complete. The output of this code fragment is as follows:

```
Creating two threads using same lock:
lock_reader(): x = 0
```

Because both threads are using the same lock, the `lock_reader()` thread cannot see any of the intermediate values of `x` produced by `lock_writer()` while holding the lock.

Quick Quiz 4.13: Is "x = 0" the only possible output from the code fragment shown in Listing 4.6? If so, why? If not, what other output could appear, and why? ■

Listing 4.7 shows a similar code fragment, but this time using different locks: `lock_a` for `lock_reader()` and `lock_b` for `lock_writer()`. The output of this code fragment is as follows:

```
Creating two threads w/different locks:
lock_reader(): x = 0
lock_reader(): x = 1
lock_reader(): x = 2
lock_reader(): x = 3
```

Because the two threads are using different locks, they do not exclude each other, and can run concurrently. The `lock_reader()` function can therefore see the intermediate values of `x` stored by `lock_writer()`.

Quick Quiz 4.14: Using different locks could cause quite a bit of confusion, what with threads seeing each others' intermediate states. So should well-written parallel programs

www.dbooks.org

restrict themselves to using a single lock in order to avoid this kind of confusion? ■

Quick Quiz 4.15: In the code shown in Listing 4.7, is `lock_reader()` guaranteed to see all the values produced by `lock_writer()`? Why or why not? ■

Quick Quiz 4.16: Wait a minute here!!! Listing 4.6 didn't initialize shared variable `x`, so why does it need to be initialized in Listing 4.7? ■

Although there is quite a bit more to POSIX exclusive locking, these primitives provide a good start and are in fact sufficient in a great many situations. The next section takes a brief look at POSIX reader-writer locking.

4.2.4 POSIX Reader-Writer Locking

The POSIX API provides a reader-writer lock, which is represented by a `pthread_rwlock_t`. As with `pthread_mutex_t`, `pthread_rwlock_t` may be statically initialized via `PTHREAD_RWLOCK_INITIALIZER` or dynamically initialized via the `pthread_rwlock_init()` primitive. The `pthread_rwlock_rdlock()` primitive read-acquires the specified `pthread_rwlock_t`, the `pthread_rwlock_wrlock()` primitive write-acquires it, and the `pthread_rwlock_unlock()` primitive releases it. Only a single thread may write-hold a given `pthread_rwlock_t` at any given time, but multiple threads may read-hold a given `pthread_rwlock_t`, at least while there is no thread currently write-holding it.

As you might expect, reader-writer locks are designed for read-mostly situations. In these situations, a reader-writer lock can provide greater scalability than can an exclusive lock because the exclusive lock is by definition limited to a single thread holding the lock at any given time, while the reader-writer lock permits an arbitrarily large number of readers to concurrently hold the lock. However, in practice, we need to know how much additional scalability is provided by reader-writer locks.

Listing 4.8 (`rwlockscale.c`) shows one way of measuring reader-writer lock scalability. Line 1 shows the definition and initialization of the reader-writer lock, line 2 shows the `holdtime` argument controlling the time each thread holds the reader-writer lock, line 3 shows the `thinktime` argument controlling the time between the release of the reader-writer lock and the next acquisition, line 4 defines the `readcounts` array into which each reader thread places the number of times it acquired the lock, and line 5 defines the `nreadersrunning` variable,

Listing 4.8: Measuring Reader-Writer Lock Scalability

```
1  pthread_rwlock_t rwl = PTHREAD_RWLOCK_INITIALIZER;
2  unsigned long holdtime = 0;
3  unsigned long thinktime = 0;
4  long long *readcounts;
5  int nreadersrunning = 0;
6
7  #define GOFLAG_INIT 0
8  #define GOFLAG_RUN  1
9  #define GOFLAG_STOP 2
10 char goflag = GOFLAG_INIT;
11
12 void *reader(void *arg)
13 {
14   int en;
15   int i;
16   long long loopcnt = 0;
17   long me = (long)arg;
18
19   __sync_fetch_and_add(&nreadersrunning, 1);
20   while (READ_ONCE(goflag) == GOFLAG_INIT) {
21     continue;
22   }
23   while (READ_ONCE(goflag) == GOFLAG_RUN) {
24     if ((en = pthread_rwlock_rdlock(&rwl)) != 0) {
25       fprintf(stderr,
26               "pthread_rwlock_rdlock: %s\n", strerror(en));
27       exit(EXIT_FAILURE);
28     }
29     for (i = 1; i < holdtime; i++) {
30       wait_microseconds(1);
31     }
32     if ((en = pthread_rwlock_unlock(&rwl)) != 0) {
33       fprintf(stderr,
34         "pthread_rwlock_unlock: %s\n", strerror(en));
35       exit(EXIT_FAILURE);
36     }
37     for (i = 1; i < thinktime; i++) {
38       wait_microseconds(1);
39     }
40     loopcnt++;
41   }
42   readcounts[me] = loopcnt;
43   return NULL;
44 }
```

which determines when all reader threads have started running.

Lines 7–10 define `goflag`, which synchronizes the start and the end of the test. This variable is initially set to `GOFLAG_INIT`, then set to `GOFLAG_RUN` after all the reader threads have started, and finally set to `GOFLAG_STOP` to terminate the test run.

Lines 12–44 define `reader()`, which is the reader thread. Line 19 atomically increments the `nreadersrunning` variable to indicate that this thread is now running, and lines 20–22 wait for the test to start. The `READ_ONCE()` primitive forces the compiler to fetch `goflag` on each pass through the loop—the compiler would otherwise be within its rights to assume that the value of `goflag` would never change.

Quick Quiz 4.17: Instead of using `READ_ONCE()` everywhere, why not just declare `goflag` as `volatile` on line 10 of Listing 4.8? ■

Quick Quiz 4.18: `READ_ONCE()` only affects the compiler, not the CPU. Don't we also need memory barriers to make sure that the change in `goflag`'s value propagates to the CPU in a timely fashion in Listing 4.8? ■

Quick Quiz 4.19: Would it ever be necessary to use `READ_ONCE()` when accessing a per-thread variable, for example, a variable declared using GCC's `__thread` storage class? ■

The loop spanning lines 23–41 carries out the performance test. Lines 24–28 acquire the lock, lines 29–31 hold the lock for the specified number of microseconds, lines 32–36 release the lock, and lines 37–39 wait for the specified number of microseconds before re-acquiring the lock. Line 40 counts this lock acquisition.

Line 42 moves the lock-acquisition count to this thread's element of the `readcounts[]` array, and line 43 returns, terminating this thread.

Figure 4.2 shows the results of running this test on a 224-core Xeon system with two hardware threads per core for a total of 448 software-visible CPUs. The `thinktime` parameter was zero for all these tests, and the `holdtime` parameter set to values ranging from one microsecond ("1us" on the graph) to 10,000 microseconds ("10000us" on the graph). The actual value plotted is:

$$\frac{L_N}{N L_1} \tag{4.1}$$

where N is the number of threads, L_N is the number of lock acquisitions by N threads, and L_1 is the number of lock acquisitions by a single thread. Given ideal hardware and software scalability, this value will always be 1.0.

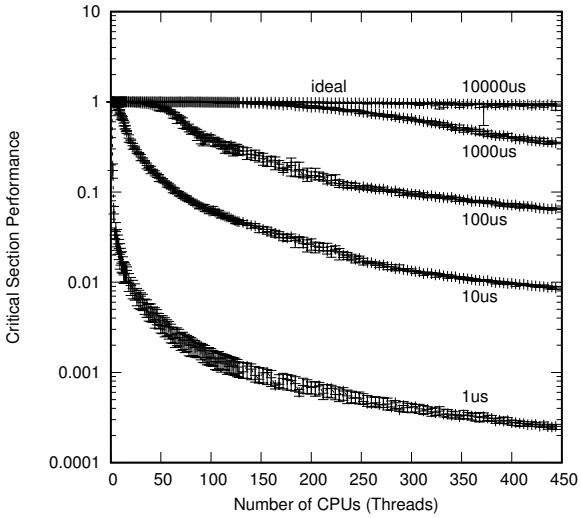


Figure 4.2: Reader-Writer Lock Scalability vs. Microseconds in Critical Section on 8-Socket System With Intel Xeon Platinum 8176 CPUs @ 2.10GHz

As can be seen in the figure, reader-writer locking scalability is decidedly non-ideal, especially for smaller sizes of critical sections. To see why read-acquisition can be so slow, consider that all the acquiring threads must update the `pthread_rwlock_t` data structure. Therefore, if all 448 executing threads attempt to read-acquire the reader-writer lock concurrently, they must update this underlying `pthread_rwlock_t` one at a time. One lucky thread might do so almost immediately, but the least-lucky thread must wait for all the other 447 threads to do their updates. This situation will only get worse as you add CPUs. Note also the logscale y-axis. Even though the 10,000 microsecond trace appears quite ideal, it has in fact degraded by about 10 % from ideal.

Quick Quiz 4.20: Isn't comparing against single-CPU throughput a bit harsh? ■

Quick Quiz 4.21: But one microsecond is not a particularly small size for a critical section. What do I do if I need a much smaller critical section, for example, one containing only a few instructions? ■

Quick Quiz 4.22: The system used is a few years old, and new hardware should be faster. So why should anyone worry about reader-writer locks being slow? ■

www.dbooks.org

Despite these limitations, reader-writer locking is quite useful in many cases, for example when the readers must do high-latency file or network I/O. There are alternatives, some of which will be presented in Chapters 5 and 9.

4.2.5 Atomic Operations (GCC Classic)

Figure 4.2 shows that the overhead of reader-writer locking is most severe for the smallest critical sections, so it would be nice to have some other way of protecting tiny critical sections. One such way uses atomic operations. We have seen an atomic operation already, namely the `__sync_fetch_and_add()` primitive on line 19 of Listing 4.8. This primitive atomically adds the value of its second argument to the value referenced by its first argument, returning the old value (which was ignored in this case). If a pair of threads concurrently execute `__sync_fetch_and_add()` on the same variable, the resulting value of the variable will include the result of both additions.

The GNU C compiler offers a number of additional atomic operations, including `__sync_fetch_and_sub()`, `__sync_fetch_and_or()`, `__sync_fetch_and_and()`, `__sync_fetch_and_xor()`, and `__sync_fetch_and_nand()`, all of which return the old value. If you instead need the new value, you can instead use the `__sync_add_and_fetch()`, `__sync_sub_and_fetch()`, `__sync_or_and_fetch()`, `__sync_and_and_fetch()`, `__sync_xor_and_fetch()`, and `__sync_nand_and_fetch()` primitives.

Quick Quiz 4.23: Is it really necessary to have both sets of primitives? ■

The classic compare-and-swap operation is provided by a pair of primitives, `__sync_bool_compare_and_swap()` and `__sync_val_compare_and_swap()`. Both of these primitive atomically update a location to a new value, but only if its prior value was equal to the specified old value. The first variant returns 1 if the operation succeeded and 0 if it failed, for example, if the prior value was not equal to the specified old value. The second variant returns the prior value of the location, which, if equal to the specified old value, indicates that the operation succeeded. Either of the compare-and-swap operation is "universal" in the sense that any atomic operation on a single location can be implemented in terms of compare-and-swap, though the earlier operations are often more efficient where they apply. The compare-and-swap operation is also capable of serving as the basis for a wider set of atomic operations, though the more elaborate of these often suffer from complexity, scalability, and performance problems [Her90].

Listing 4.9: Compiler Barrier Primitive (for GCC)

```
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))
#define READ_ONCE(x) \
                ({ typeof(x) ___x = ACCESS_ONCE(x); ___x; })
#define WRITE_ONCE(x, val) \
                do { ACCESS_ONCE(x) = (val); } while (0)
#define barrier() __asm__ __volatile__("": : :"memory")
```

Quick Quiz 4.24: Given that these atomic operations will often be able to generate single atomic instructions that are directly supported by the underlying instruction set, shouldn't they be the fastest possible way to get things done? ■

The `__sync_synchronize()` primitive issues a "memory barrier", which constrains both the compiler's and the CPU's ability to reorder operations, as discussed in Chapter 15. In some cases, it is sufficient to constrain the compiler's ability to reorder operations, while allowing the CPU free rein, in which case the `barrier()` primitive may be used. In some cases, it is only necessary to ensure that the compiler avoids optimizing away a given memory read, in which case the `READ_ONCE()` primitive may be used, as it was on line 20 of Listing 4.5. Similarly, the `WRITE_ONCE()` primitive may be used to prevent the compiler from optimizing away a given memory write. These last three primitives are not provided directly by GCC, but may be implemented straightforwardly as shown in Listing 4.9, and all three are discussed at length in Section 4.3.4. Alternatively, `READ_ONCE(x)` has much in common with the GCC intrinsic `__atomic_load_n(&x, __ATOMIC_RELAXED)` and `WRITE_ONCE()` has much in common with the GCC intrinsic `__atomic_store_n(&x, v, __ATOMIC_RELAXED)`.

Quick Quiz 4.25: What happened to `ACCESS_ONCE()`? ■

4.2.6 Atomic Operations (C11)

The C11 standard added atomic operations, including loads (`atomic_load()`), stores (`atomic_store()`), memory barriers (`atomic_thread_fence()` and `atomic_signal_fence()`), and read-modify-write atomics. The read-modify-write atomics include `atomic_fetch_add()`, `atomic_fetch_sub()`, `atomic_fetch_and()`, `atomic_fetch_xor()`, `atomic_exchange()`, `atomic_compare_exchange_strong()`, and `atomic_compare_exchange_weak()`. These operate in a manner similar to those described

in Section 4.2.5, but with the addition of memory-order arguments to `_explicit` variants of all of the operations. Without memory-order arguments, all the atomic operations are fully ordered, and the arguments permit weaker orderings. For example, "`atomic_load_explicit(&a, memory_order_relaxed)`" is vaguely similar to the Linux kernel's "`READ_ONCE()`".[1]

4.2.7 Atomic Operations (Modern GCC)

One restriction of the C11 atomics is that they apply only to special atomic types, which can be problematic. The GNU C compiler therefore provides atomic intrinsics, including `__atomic_load()`, `__atomic_load_n()`, `__atomic_store()`, `__atomic_store_n()`, `__atomic_thread_fence()`, etc. These intrinsics offer the same semantics as their C11 counterparts, but may be used on plain non-atomic objects. Some of these intrinsics may be passed a memory-order argument from this list: `__ATOMIC_RELAXED`, `__ATOMIC_CONSUME`, `__ATOMIC_ACQUIRE`, `__ATOMIC_RELEASE`, `__ATOMIC_ACQ_REL`, and `__ATOMIC_SEQ_CST`.

4.2.8 Per-Thread Variables

Per-thread variables, also called thread-specific data, thread-local storage, and other less-polite names, are used extremely heavily in concurrent code, as will be explored in Chapters 5 and 8. POSIX supplies the `pthread_key_create()` function to create a per-thread variable (and return the corresponding key), `pthread_key_delete()` to delete the per-thread variable corresponding to key, `pthread_setspecific()` to set the value of the current thread's variable corresponding to the specified key, and `pthread_getspecific()` to return that value.

A number of compilers (including GCC) provide a `__thread` specifier that may be used in a variable definition to designate that variable as being per-thread. The name of the variable may then be used normally to access the value of the current thread's instance of that variable. Of course, `__thread` is much easier to use than the POSIX thead-specific data, and so `__thread` is usually preferred for code that is to be built only with GCC or other compilers supporting `__thread`.

Fortunately, the C11 standard introduced a `_Thread_local` keyword that can be used in place of `__thread`. In the fullness of time, this new keyword should combine the ease of use of `__thread` with the portability of POSIX thread-specific data.

4.3 Alternatives to POSIX Operations

> The strategic marketing paradigm of Open Source is a massively parallel drunkard's walk filtered by a Darwinistic process.
>
> *Bruce Perens*

Unfortunately, threading operations, locking primitives, and atomic operations were in reasonably wide use long before the various standards committees got around to them. As a result, there is considerable variation in how these operations are supported. It is still quite common to find these operations implemented in assembly language, either for historical reasons or to obtain better performance in specialized circumstances. For example, GCC's `__sync_` family of primitives all provide full memory-ordering semantics, which in the past motivated many developers to create their own implementations for situations where the full memory ordering semantics are not required. The following sections show some alternatives from the Linux kernel and some historical primitives used by this book's sample code.

4.3.1 Organization and Initialization

Although many environments do not require any special initialization code, the code samples in this book start with a call to `smp_init()`, which initializes a mapping from `pthread_t` to consecutive integers. The userspace RCU library[2] similarly requires a call to `rcu_init()`. Although these calls can be hidden in environments (such as that of GCC) that support constructors, most of the RCU flavors supported by the userspace RCU library also require each thread invoke `rcu_register_thread()` upon thread creation and `rcu_unregister_thread()` before thread exit.

In the case of the Linux kernel, it is a philosophical question as to whether the kernel does not require calls to special initialization code or whether the kernel's boot-time code is in fact the required initialization code.

[1] Memory ordering is described in more detail in Chapter 15 and Appendix C.

[2] See Section 9.5 for more information on RCU.

www.dbooks.org

Listing 4.10: Thread API

```
int smp_thread_id(void)
thread_id_t create_thread(void *(*func)(void *), void *arg)
for_each_thread(t)
for_each_running_thread(t)
void *wait_thread(thread_id_t tid)
void wait_all_threads(void)
```

4.3.2 Thread Creation, Destruction, and Control

The Linux kernel uses `struct task_struct` pointers to track kthreads, `kthread_create()` to create them, `kthread_should_stop()` to externally suggest that they stop (which has no POSIX equivalent),[3] `kthread_stop()` to wait for them to stop, and `schedule_timeout_interruptible()` for a timed wait. There are quite a few additional kthread-management APIs, but this provides a good start, as well as good search terms.

The CodeSamples API focuses on "threads", which are a locus of control.[4] Each such thread has an identifier of type `thread_id_t`, and no two threads running at a given time will have the same identifier. Threads share everything except for per-thread local state,[5] which includes program counter and stack.

The thread API is shown in Listing 4.10, and members are described in the following sections.

4.3.2.1 `create_thread()`

The `create_thread()` primitive creates a new thread, starting the new thread's execution at the function `func` specified by `create_thread()`'s first argument, and passing it the argument specified by `create_thread()`'s second argument. This newly created thread will terminate when it returns from the starting function specified by `func`. The `create_thread()` primitive returns the `thread_id_t` corresponding to the newly created child thread.

This primitive will abort the program if more than `NR_THREADS` threads are created, counting the one implicitly created by running the program. `NR_THREADS` is a compile-time constant that may be modified, though some systems may have an upper bound for the allowable number of threads.

[3] POSIX environments can work around the lack of `kthread_should_stop()` by using a properly synchronized boolean flag in conjunction with `pthread_join()`.

[4] There are many other names for similar software constructs, including "process", "task", "fiber", "event", "execution agent", and so on. Similar design principles apply to all of them.

[5] How is that for a circular definition?

4.3.2.2 `smp_thread_id()`

Because the `thread_id_t` returned from `create_thread()` is system-dependent, the `smp_thread_id()` primitive returns a thread index corresponding to the thread making the request. This index is guaranteed to be less than the maximum number of threads that have been in existence since the program started, and is therefore useful for bitmasks, array indices, and the like.

4.3.2.3 `for_each_thread()`

The `for_each_thread()` macro loops through all threads that exist, including all threads that *would* exist if created. This macro is useful for handling per-thread variables as will be seen in Section 4.2.8.

4.3.2.4 `for_each_running_thread()`

The `for_each_running_thread()` macro loops through only those threads that currently exist. It is the caller's responsibility to synchronize with thread creation and deletion if required.

4.3.2.5 `wait_thread()`

The `wait_thread()` primitive waits for completion of the thread specified by the `thread_id_t` passed to it. This in no way interferes with the execution of the specified thread; instead, it merely waits for it. Note that `wait_thread()` returns the value that was returned by the corresponding thread.

4.3.2.6 `wait_all_threads()`

The `wait_all_threads()` primitive waits for completion of all currently running threads. It is the caller's responsibility to synchronize with thread creation and deletion if required. However, this primitive is normally used to clean up at the end of a run, so such synchronization is normally not needed.

4.3.2.7 Example Usage

Listing 4.11 (`threadcreate.c`) shows an example hello-world-like child thread. As noted earlier, each thread is allocated its own stack, so each thread has its own private `arg` argument and `myarg` variable. Each child simply prints its argument and its `smp_thread_id()` before exiting. Note that the `return` statement on line 7 terminates the thread, returning a `NULL` to whoever invokes `wait_thread()` on this thread.

Listing 4.11: Example Child Thread

```
1 void *thread_test(void *arg)
2 {
3   int myarg = (intptr_t)arg;
4
5   printf("child thread %d: smp_thread_id() = %d\n",
6          myarg, smp_thread_id());
7   return NULL;
8 }
```

Listing 4.12: Example Parent Thread

```
1 int main(int argc, char *argv[])
2 {
3   int i;
4   int nkids = 1;
5
6   smp_init();
7
8   if (argc > 1) {
9     nkids = strtoul(argv[1], NULL, 0);
10    if (nkids > NR_THREADS) {
11      fprintf(stderr, "nkids = %d too large, max = %d\n",
12        nkids, NR_THREADS);
13      usage(argv[0]);
14    }
15  }
16  printf("Parent thread spawning %d threads.\n", nkids);
17
18  for (i = 0; i < nkids; i++)
19    create_thread(thread_test, (void *)(intptr_t)i);
20
21  wait_all_threads();
22
23  printf("All spawned threads completed.\n");
24
25  exit(0);
26 }
```

The parent program is shown in Listing 4.12. It invokes `smp_init()` to initialize the threading system on line 6, parses arguments on lines 8–15, and announces its presence on line 16. It creates the specified number of child threads on lines 18–19, and waits for them to complete on line 21. Note that `wait_all_threads()` discards the threads return values, as in this case they are all `NULL`, which is not very interesting.

Quick Quiz 4.26: What happened to the Linux-kernel equivalents to `fork()` and `wait()`? ■

4.3.3 Locking

A good starting subset of the Linux kernel's locking API is shown in Listing 4.13, each API element being described in the following sections. This book's CodeSamples locking API closely follows that of the Linux kernel.

Listing 4.13: Locking API

```
void spin_lock_init(spinlock_t *sp);
void spin_lock(spinlock_t *sp);
int spin_trylock(spinlock_t *sp);
void spin_unlock(spinlock_t *sp);
```

4.3.3.1 `spin_lock_init()`

The `spin_lock_init()` primitive initializes the specified `spinlock_t` variable, and must be invoked before this variable is passed to any other spinlock primitive.

4.3.3.2 `spin_lock()`

The `spin_lock()` primitive acquires the specified spinlock, if necessary, waiting until the spinlock becomes available. In some environments, such as pthreads, this waiting will involve blocking, while in others, such as the Linux kernel, it might involve a CPU-bound spin loop.

The key point is that only one thread may hold a spinlock at any given time.

4.3.3.3 `spin_trylock()`

The `spin_trylock()` primitive acquires the specified spinlock, but only if it is immediately available. It returns `true` if it was able to acquire the spinlock and `false` otherwise.

4.3.3.4 `spin_unlock()`

The `spin_unlock()` primitive releases the specified spinlock, allowing other threads to acquire it.

4.3.3.5 Example Usage

A spinlock named `mutex` may be used to protect a variable `counter` as follows:

```
spin_lock(&mutex);
counter++;
spin_unlock(&mutex);
```

Quick Quiz 4.27: What problems could occur if the variable `counter` were incremented without the protection of `mutex`? ■

However, the `spin_lock()` and `spin_unlock()` primitives do have performance consequences, as will be seen in Chapter 10.

www.dbooks.org

Listing 4.14: Living Dangerously Early 1990s Style

```
1 ptr = global_ptr;
2 if (ptr != NULL && ptr < high_address)
3   do_low(ptr);
```

Listing 4.15: C Compilers Can Invent Loads

```
1 if (global_ptr != NULL &&
2     global_ptr < high_address)
3   do_low(global_ptr);
```

4.3.4 Accessing Shared Variables

It was not until 2011 that the C standard defined semantics for concurrent read/write access to shared variables. However, concurrent C code was being written at least a quarter century earlier [BK85, Inm85]. This raises the question as to what today's greybeards did back in long-past pre-C11 days. A short answer to this question is "they lived dangerously".

At least they would have been living dangerously had they been using 2018 compilers. In (say) the early 1990s, compilers did fewer optimizations, in part because there were fewer compiler writers and in part due to the relatively small memories of that era. Nevertheless, problems did arise, as shown in Listing 4.14, which the compiler is within its rights to transform into Listing 4.15. As you can, the temporary on line 1 of Listing 4.14 has been optimized away, so that `global_ptr` will be loaded up to three times.

Quick Quiz 4.28: What is wrong with loading Listing 4.14's `global_ptr` up to three times? ■

Section 4.3.4.1 describes additional problems caused by plain accesses, Sections 4.3.4.2 and 4.3.4.3 describe some pre-C11 solutions. Of course, where practical, the primitives described in Section 4.2.5 or (especially) Section 4.2.6 should instead be used to avoid data races, that is, to ensure that if there are multiple concurrent accesses to a given variable, all of those accesses are loads.

4.3.4.1 Shared-Variable Shenanigans

Given code that does plain loads and stores,[6] the compiler is within its rights to assume that the affected variables are neither accessed nor modified by any other thread. This assumption allows the compiler to carry out a large number of transformations, including load tearing, store tearing, load fusing, store fusing, code reordering, invented loads, invented stores, store-to-load transformations, and dead-code elimination, all of which work just fine in single-threaded code. But concurrent code can be broken by each of these transformations, or shared-variable shenanigans, as described below.

Load tearing occurs when the compiler uses multiple load instructions for a single access. For example, the compiler could in theory compile the load from `global_ptr` (see line 1 of Listing 4.14) as a series of one-byte loads. If some other thread was concurrently setting `global_ptr` to `NULL`, the result might have all but one byte of the pointer set to zero, thus forming a "wild pointer". Stores using such a wild pointer could corrupt arbitrary regions of memory, resulting in rare and difficult-to-debug crashes.

Worse yet, on (say) an 8-bit system with 16-bit pointers, the compiler might have no choice but to use a pair of 8-bit instructions to access a given pointer. Because the C standard must support all manner of systems, the standard cannot rule out load tearing in the general case.

Store tearing occurs when the compiler uses multiple store instructions for a single access. For example, one thread might store `0x12345678` to a four-byte integer variable at the same time another thread stored `0xabcdef00`. If the compiler used 16-bit stores for either access, the result might well be `0x1234ef00`, which could come as quite a surprise to code loading from this integer. Nor is this a strictly theoretical issue. For example, there are CPUs that feature small immediate instruction fields, and on such CPUs, the compiler might split a 64-bit store into two 32-bit stores in order to reduce the overhead of explicitly forming the 64-bit constant in a register, even on a 64-bit CPU. There are historical reports of this actually happening in the wild (e.g. [KM13]), but there is also a recent report [Dea19].[7]

Of course, the compiler simply has no choice but to tear some stores in the general case, given the possibility of code using 64-bit integers running on a 32-bit system. But for properly aligned machine-sized stores, `WRITE_ONCE()` will prevent store tearing.

Load fusing occurs when the compiler uses the result of a prior load from a given variable instead of repeating the load. Not only is this sort of optimization just fine in single-threaded code, it is often just fine in multithreaded

[6] That is, normal loads and stores instead of C11 atomics, inline assembly, or volatile accesses.

[7] Note that this tearing can happen even on properly aligned and machine-word-sized accesses, and in this particular case, even for volatile stores. Some might argue that this behavior constitutes a bug in the compiler, but either way it illustrates the perceived value of store tearing from a compiler-writer viewpoint.

Listing 4.16: Inviting Load Fusing

```
1 while (!need_to_stop)
2   do_something_quickly();
```

Listing 4.17: C Compilers Can Fuse Loads

```
1 if (!need_to_stop)
2   for (;;) {
3     do_something_quickly();
4     do_something_quickly();
5     do_something_quickly();
6     do_something_quickly();
7     do_something_quickly();
8     do_something_quickly();
9     do_something_quickly();
10    do_something_quickly();
11    do_something_quickly();
12    do_something_quickly();
13    do_something_quickly();
14    do_something_quickly();
15    do_something_quickly();
16    do_something_quickly();
17    do_something_quickly();
18    do_something_quickly();
19  }
```

code. Unfortunately, the word "often" hides some truly annoying exceptions.

For example, suppose that a real-time system needs to invoke a function named `do_something_quickly()` repeatedly until the variable `need_to_stop` was set, and that the compiler can see that `do_something_quickly()` does not store to `need_to_stop`. One (unsafe) way to code this is shown in Listing 4.16. The compiler might reasonably unroll this loop sixteen times in order to reduce the per-invocation of the backwards branch at the end of the loop. Worse yet, because the compiler knows that `do_something_quickly()` does not store to `need_to_stop`, the compiler could quite reasonably decide to check this variable only once, resulting in the code shown in Listing 4.17. Once entered, the loop on lines 2–19 will never exit, regardless of how many times some other thread stores a non-zero value to `need_to_stop`. The result will at best be consternation, and might well also include severe physical damage.

The compiler can fuse loads across surprisingly large spans of code. For example, in Listing 4.18, `t0()` and `t1()` run concurrently, and `do_something()` and `do_something_else()` are inline functions. Line 1 declares pointer `gp`, which C initializes to `NULL` by default. At some point, line 5 of `t0()` stores a non-`NULL` pointer to `gp`. Meanwhile, `t1()` loads from `gp` three times on lines 10, 12, and 15. Given that line 13 finds that `gp` is non-`NULL`, one might hope that the dereference on line 15 would be guaranteed never to fault. Unfortunately, the compiler is within its rights to fuse the read on lines 10 and 15, which

Listing 4.18: C Compilers Can Fuse Non-Adjacent Loads

```
1 int *gp;
2
3 void t0(void)
4 {
5   WRITE_ONCE(gp, &myvar);
6 }
7
8 void t1(void)
9 {
10  p1 = gp;
11  do_something(p1);
12  p2 = READ_ONCE(gp);
13  if (p2) {
14    do_something_else();
15    p3 = *gp;
16  }
17 }
```

Listing 4.19: C Compilers Can Fuse Stores

```
1 void shut_it_down(void)
2 {
3   status = SHUTTING_DOWN; /* BUGGY!!! */
4   start_shutdown();
5   while (!other_task_ready) /* BUGGY!!! */
6     continue;
7   finish_shutdown();
8   status = SHUT_DOWN; /* BUGGY!!! */
9   do_something_else();
10 }
11
12 void work_until_shut_down(void)
13 {
14   while (status != SHUTTING_DOWN) /* BUGGY!!! */
15     do_more_work();
16   other_task_ready = 1; /* BUGGY!!! */
17 }
```

means that if line 10 loads `NULL` and line 12 loads `&myvar`, line 15 could load `NULL`, resulting in a fault.[8] Note that the intervening `READ_ONCE()` does not prevent the other two loads from being fused, despite the fact that all three are loading from the same variable.

Quick Quiz 4.29: Why does it matter whether `do_something()` and `do_something_else()` in Listing 4.18 are inline functions? ■

Store fusing can occur when the compiler notices a pair of successive stores to a given variable with no intervening loads from that variable. In this case, the compiler is within its rights to omit the first store. This is never a problem in single-threaded code, and in fact it is usually not a problem in correctly written concurrent code. After all, if the two stores are executed in quick succession, there is very little chance that some other thread could load the value from the first store.

However, there are exceptions, for example as shown in Listing 4.19. The function `shut_it_down()` stores

[8] Will Deacon reports that this happened in the Linux kernel.

www.dbooks.org

to the shared variable status on lines 3 and 8, and so assuming that neither start_shutdown() nor finish_shutdown() access status, the compiler could reasonably remove the store to status on line 3. Unfortunately, this would mean that work_until_shut_down() would never exit its loop spanning lines 14 and 15, and thus would never set other_task_ready, which would in turn mean that shut_it_down() would never exit its loop spanning lines 5 and 6, even if the compiler chooses not to fuse the successive loads from other_task_ready on line 5.

And there are more problems with the code in Listing 4.19, including code reordering.

Code reordering is a common compilation technique used to combine common subexpressions, reduce register pressure, and improve utilization of the many functional units available on modern superscalar microprocessors. It is also another reason why the code in Listing 4.19 is buggy. For example, suppose that the do_more_work() function on line 15 does not access other_task_ready. Then the compiler would be within its rights to move the assignment to other_task_ready on line 16 to precede line 14, which might be a great disappointment for anyone hoping that the last call to do_more_work() on line 15 happens before the call to finish_shutdown() on line 7.

It might seem futile to prevent the compiler from changing the order of accesses in cases where the underlying hardware is free to reorder them. However, modern machines have *exact exceptions* and *exact interrupts*, meaning that any interrupt or exception will appear to have happened at a specific place in the instruction stream. This means that the handler will see the effect of all prior instructions, but won't see the effect of any subsequent instructions. READ_ONCE() and WRITE_ONCE() can therefore be used to control communication between interrupted code and interrupt handlers, independent of the ordering provided by the underlying hardware.[9]

Invented loads were illustrated by the code in Listings 4.14 and 4.15, in which the compiler optimized away a temporary variable, thus loading from a shared variable more often than intended.

Invented loads can be a performance hazard. These hazards can occur when a load of variable in a "hot" cacheline is hoisted out of an if statement. These hoisting optimizations are not uncommon, and can cause significant increases in cache misses, and thus significant degradation of both performance and scalability.

[9] That said, the various standards committees would prefer that you use atomics or variables of type sig_atomic_t, instead of READ_ONCE() and WRITE_ONCE().

Listing 4.20: Inviting an Invented Store

```
1 if (condition)
2   a = 1;
3 else
4   do_a_bunch_of_stuff();
```

Listing 4.21: Compiler Invents an Invited Store

```
1 a = 1;
2 if (!condition) {
3   a = 0;
4   do_a_bunch_of_stuff();
5 }
```

Invented stores can occur in a number of situations. For example, a compiler emitting code for work_until_shut_down() in Listing 4.19 might notice that other_task_ready is not accessed by do_more_work(), and stored to on line 16. If do_more_work() was a complex inline function, it might be necessary to do a register spill, in which case one attractive place to use for temporary storage is other_task_ready. After all, there are no accesses to it, so what is the harm?

Of course, a non-zero store to this variable at just the wrong time would result in the while loop on line 5 terminating prematurely, again allowing finish_shutdown() to run concurrently with do_more_work(). Given that the entire point of this while appears to be to prevent such concurrency, this is not a good thing.

Using a stored-to variable as a temporary might seem outlandish, but it is permitted by the standard. Nevertheless, readers might be justified in wanting a less outlandish example, which is provided by Listings 4.20 and 4.21.

A compiler emitting code for Listing 4.20 might know that the value of a is initially zero, which might be a strong temptation to optimize away one branch by transforming this code to that in Listing 4.21. Here, line 1 unconditionally stores 1 to a, then resets the value back to zero on line 3 if condition was not set. This transforms the if-then-else into an if-then, saving one branch.

Quick Quiz 4.30: Ouch! So can't the compiler invent a store to a normal variable pretty much any time it likes? ■

Finally, pre-C11 compilers could invent writes to unrelated variables that happened to be adjacent to written-to variables [Boe05, Section 4.2]. This variant of invented stores has been outlawed by the prohibition against compiler optimizations that invent data races.

Store-to-load transformations can occur when the compiler notices that a plain store might not actually change the value in memory. For example, consider Listing 4.22. Line 1 fetches p, but the "if" statement

Listing 4.22: Inviting a Store-to-Load Conversion

```
1 r1 = p;
2 if (unlikely(r1))
3   do_something_with(r1);
4 barrier();
5 p = NULL;
```

Listing 4.23: Compiler Converts a Store to a Load

```
1 r1 = p;
2 if (unlikely(r1))
3   do_something_with(r1);
4 barrier();
5 if (p != NULL)
6   p = NULL;
```

on line 2 also tells the compiler that the developer thinks that p is usually zero.[10] The barrier() statment on line 4 forces the compiler to forget the value of p, but one could imagine a compiler choosing to remember the hint—or getting an additional hint via feedback-directed optimization. Doing so would cause the compiler to realize that line 5 is often an expensive no-op.

Such a compiler might therefore guard the store of NULL with a check, as shown on lines 5 and 6 of Listing 4.23. Although this transformation is often desirable, it could be problematic if the actual store was required for ordering. For example, a write memory barrier (Linux kernel smp_wmb()) would order the store, but not the load. This situation might suggest use of smp_store_release() over smp_wmb().

Dead-code elimination can occur when the compiler notices that the value from a load is never used, or when a variable is stored to, but never loaded from. This can of course eliminate an access to a shared variable, which can in turn defeat a memory-ordering primitive, which could cause your concurrent code to act in surprising ways. Experience thus far indicates that relatively few such surprises will be at all pleasant. Elimination of store-only variables is especially dangerous in cases where external code locates the variable via symbol tables: The compiler is necessarily ignorant of such external-code accesses, and might thus eliminate a variable that the external code relies upon.

Reliable concurrent code clearly needs a way to cause the compiler to preserve the number, order, and type of important accesses to shared memory, a topic taken up by Sections 4.3.4.2 and 4.3.4.3, which are up next.

[10] The unlikely() function provides this hint to the compiler, and different compilers provide different ways of implementing unlikely().

4.3.4.2 A Volatile Solution

Although it is now much maligned, before the advent of C11 and C++11 [Bec11], the volatile keyword was an indispensible tool in the parallel programmer's toolbox. This raises the question of exactly what volatile means, a question that is not answered with excessive precision even by more recent versions of this standard [Smi19].[11] This version guarantees that "Accesses through volatile glvalues are evaluated strictly according to the rules of the abstract machine", that volatile accesses are side effects, that they are one of the four forward-progress indicators, and that their exact semantics are implementation-defined. Perhaps the clearest guidance is provided by this non-normative note:

> volatile is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. Furthermore, for some implementations, volatile might indicate that special hardware instructions are required to access the object. See 6.8.1 for detailed semantics. In general, the semantics of volatile are intended to be the same in C++ as they are in C.

This wording might be reassuring to those writing low-level code, except for the fact that compiler writers are free to completely ignore non-normative notes. Parallel programmers might instead reassure themselves that compiler writers would like to avoid breaking device drivers (though perhaps only after a few "frank and open" discussions with device-driver developers), and device drivers impose at least the following constraints [MWPF18]:

1. Implementations are forbidden from tearing an aligned volatile access when machine instructions of that access's size and type are available.[12] Concurrent code relies on this constraint to avoid unnecessary load and store tearing.

2. Implementations must not assume anything about the semantics of a volatile access, nor, for any volatile access that returns a value, about the possible set of values that might be returned.[13] Concurrent code

[11] JF Bastien thoroughly documented the history and use cases for the volatile keyword in C++ [Bas18].

[12] Note that this leaves unspecified what to do with 128-bit loads and stores on CPUs having 128-bit CAS but not 128-bit loads and stores.

[13] This is strongly implied by the implementation-defined semantics called out above.

www.dbooks.org

Listing 4.24: Avoiding Danger, 2018 Style

```
1 ptr = READ_ONCE(global_ptr);
2 if (ptr != NULL && ptr < high_address)
3   do_low(ptr);
```

Listing 4.25: Preventing Load Fusing

```
1 while (!READ_ONCE(need_to_stop))
2   do_something_quickly();
```

Listing 4.26: Preventing Store Fusing and Invented Stores

```
1 void shut_it_down(void)
2 {
3   WRITE_ONCE(status, SHUTTING_DOWN); /* BUGGY!!! */
4   start_shutdown();
5   while (!READ_ONCE(other_task_ready)) /* BUGGY!!! */
6     continue;
7   finish_shutdown();
8   WRITE_ONCE(status, SHUT_DOWN); /* BUGGY!!! */
9   do_something_else();
10 }
11
12 void work_until_shut_down(void)
13 {
14   while (READ_ONCE(status) != SHUTTING_DOWN) /* BUGGY!!! */
15     do_more_work();
16   WRITE_ONCE(other_task_ready, 1); /* BUGGY!!! */
17 }
```

relies on this constraint to avoid optimizations that are inapplicable given that other processors might be concurrently accessing the location in question.

3. Aligned machine-sized non-mixed-size volatile accesses interact naturally with volatile assembly-code sequences before and after. This is necessary because some devices must be accessed using a combination of volatile MMIO accesses and special-purpose assembly-language instructions. Concurrent code relies on this constraint in order to achieve the desired ordering properties from combinations of volatile accesses and other means discussed in Section 4.3.4.3.

Concurrent code also relies on the first two constraints to avoid undefined behavior that could result due to data races if any of the accesses to a given object was either non-atomic or non-volatile, assuming that all accesses are aligned and machine-sized. The semantics of mixed-size accesses to the same locations are more complex, and are left aside for the time being.

So how does `volatile` stack up against the earlier examples?

Using `READ_ONCE()` on line 1 of Listing 4.14 avoids invented loads, resulting in the code shown in Listing 4.24.

As shown in Listing 4.25, `READ_ONCE()` can also prevent the loop unrolling in Listing 4.17.

Listing 4.27: Disinviting an Invented Store

```
1 if (condition)
2   WRITE_ONCE(a, 1);
3 else
4   do_a_bunch_of_stuff();
```

Listing 4.28: Preventing C Compilers From Fusing Loads

```
1 while (!need_to_stop) {
2   barrier();
3   do_something_quickly();
4   barrier();
5 }
```

`READ_ONCE()` and `WRITE_ONCE()` can also be used to prevent the store fusing and invented stores that were shown in Listing 4.19, with the result shown in Listing 4.26. However, this does nothing to prevent code reordering, which requires some additional tricks taught in Section 4.3.4.3.

Finally, `WRITE_ONCE()` can be used to prevent the store invention shown in Listing 4.20, with the resulting code shown in Listing 4.27.

To summarize, the `volatile` keyword can prevent load tearing and store tearing in cases where the loads and stores are machine-sized and properly aligned. It can also prevent load fusing, store fusing, invented loads, and invented stores. However, although it does prevent the compiler from reordering `volatile` accesses with each other, it does nothing to prevent the CPU from reordering these accesses. Furthermore, it does nothing to prevent either compiler or CPU from reordering non-`volatile` accesses with each other or with `volatile` accesses. Preventing these types of reordering requires the techniques described in the next section.

4.3.4.3 Assembling the Rest of a Solution

Additional ordering has traditionally been provided by recourse to assembly language, for example, GCC asm directives. Oddly enough, these directives need not actually contain assembly language, as exemplified by the `barrier()` macro shown in Listing 4.9.

In the `barrier()` macro, the `__asm__` introduces the asm directive, the `__volatile__` prevents the compiler from optimizing the asm away, the empty string specifies that no actual instructions are to be emitted, and the final `"memory"` tells the compiler that this do-nothing asm can arbitrarily change memory. In response, the compiler will avoid moving any memory references across the `barrier()` macro. This means that the real-time-destroying loop unrolling shown in Listing 4.17 can be

Listing 4.29: Preventing Reordering

```
1  void shut_it_down(void)
2  {
3    WRITE_ONCE(status, SHUTTING_DOWN);
4    smp_mb();
5    start_shutdown();
6    while (!READ_ONCE(other_task_ready))
7      continue;
8    smp_mb();
9    finish_shutdown();
10   smp_mb();
11   WRITE_ONCE(status, SHUT_DOWN);
12   do_something_else();
13 }
14
15 void work_until_shut_down(void)
16 {
17   while (READ_ONCE(status) != SHUTTING_DOWN) {
18     smp_mb();
19     do_more_work();
20   }
21   smp_mb();
22   WRITE_ONCE(other_task_ready, 1);
23 }
```

prevented by adding `barrier()` calls as shown on lines 2 and 4 of Listing 4.28. These two lines of code prevent the compiler from pushing the load from `need_to_stop` into or past `do_something_quickly()` from either direction.

However, this does nothing to prevent the CPU from reordering the references. In many cases, this is not a problem because the hardware can only do a certain amount of reordering. However, there are cases such as Listing 4.19 where the hardware must be constrained. Listing 4.26 prevented store fusing and invention, and Listing 4.29 further prevents the remaining reordering by addition of `smp_mb()` on lines 4, 8, 10, 18, and 21. The `smp_mb()` macro is similar to `barrier()` shown in Listing 4.9, but with the empty string replaced by a string containing the instruction for a full memory barrier, for example, `"mfence"` on x86 or `"sync"` on PowerPC.

Quick Quiz 4.31: But aren't full memory barriers very heavyweight? Isn't there a cheaper way to enforce the ordering needed in Listing 4.29? ■

Ordering is also provided by some read-modify-write atomic operations, some of which are presented in Section 4.3.5. In the general case, memory ordering can be quite subtle, as discussed in Chapter 15. The next section covers an alternative to memory ordering, namely limiting or even entirely avoiding data races.

4.3.4.4 Avoiding Data Races

> "Doctor, it hurts my head when I think about concurrently accessing shared variables!"
>
> "Then stop concurrently accessing shared variables!!!"

The doctor's advice might seem unhelpful, but one time-tested way to avoid concurrently accessing shared variables is access those variables only when holding a particular lock, as will be discussed in Chapter 7. Another way is to access a given "shared" variable only from a given CPU or thread, as will be discussed in Chapter 8. It is possible to combine these two approaches, for example, a given variable might be modified only by a given CPU or thread while holding a particular lock, and might be read either from that same CPU or thread on the one hand, or from some other CPU or thread while holding that same lock on the other. In all of these situations, all accesses to the shared variables may be plain C-language accesses.

Here is a list of situations allowing plain loads and stores for some accesses to a given variable, while requiring markings (such as `READ_ONCE()` and `WRITE_ONCE()`) for other accesses to that same variable:

1. A shared variable is only modified by a given owning CPU or thread, but is read by other CPUs or threads. All stores must use `WRITE_ONCE()`. The owning CPU or thread may use plain loads. Everything else must use `READ_ONCE()` for loads.

2. A shared variable is only modified while holding a given lock, but is read by code not holding that lock. All stores must use `WRITE_ONCE()`. CPUs or threads holding the lock may use plain loads. Everything else must use `READ_ONCE()` for loads.

3. A shared variable is only modified while holding a given lock by a given owning CPU or thread, but is read by other CPUs or threads or by code not holding that lock. All stores must use `WRITE_ONCE()`. The owning CPU or thread may use plain loads, as may any CPU or thread holding the lock. Everything else must use `READ_ONCE()` for loads.

4. A shared variable is only accessed by a given CPU or thread and by a signal or interrupt handler running in that CPU's or thread's context. The handler can use plain loads and stores, as can any code that has prevented the handler from being invoked, that is, code that has blocked signals and/or interrupts. All other code must use `READ_ONCE()` and `WRITE_ONCE()`.

5. A shared variable is only accessed by a given CPU or thread and by a signal or interrupt handler running

www.dbooks.org

in that CPU's or thread's context, and the handler always restores the values of any variables that it has written before return. The handler can use plain loads and stores, as can any code that has prevented the handler from being invoked, that is, code that has blocked signals and/or interrupts. All other code can use plain loads, but must use `WRITE_ONCE()` to prevent store tearing, store fusing, and invented stores.

Quick Quiz 4.32: What needs to happen if an interrupt or signal handler might itself be interrupted? ■

In most other cases, loads from and stores to a shared variable must use `READ_ONCE()` and `WRITE_ONCE()` or stronger, respectively. But it bears repeating that neither `READ_ONCE()` nor `WRITE_ONCE()` provide any ordering guarantees other than within the compiler. See the above Section 4.3.4.3 or Chapter 15 for information on such guarantees.

Examples of many of these data-race-avoidance patterns are presented in Chapter 5.

4.3.5 Atomic Operations

The Linux kernel provides a wide variety of atomic operations, but those defined on type `atomic_t` provide a good start. Normal non-tearing reads and stores are provided by `atomic_read()` and `atomic_set()`, respectively. Acquire load is provided by `smp_load_acquire()` and release store by `smp_store_release()`.

Non-value-returning fetch-and-add operations are provided by `atomic_add()`, `atomic_sub()`, `atomic_inc()`, and `atomic_dec()`, among others. An atomic decrement that returns a reached-zero indication is provided by both `atomic_dec_and_test()` and `atomic_sub_and_test()`. An atomic add that returns the new value is provided by `atomic_add_return()`. Both `atomic_add_unless()` and `atomic_inc_not_zero()` provide conditional atomic operations, where nothing happens unless the original value of the atomic variable is different than the value specified (these are very handy for managing reference counters, for example).

An atomic exchange operation is provided by `atomic_xchg()`, and the celebrated compare-and-swap (CAS) operation is provided by `atomic_cmpxchg()`. Both of these return the old value. Many additional atomic RMW primitives are available in the Linux kernel, see the `Documentation/core-api/atomic_ops.rst` file in the Linux-kernel source tree.

Listing 4.30: Per-Thread-Variable API

```
DEFINE_PER_THREAD(type, name)
DECLARE_PER_THREAD(type, name)
per_thread(name, thread)
__get_thread_var(name)
init_per_thread(name, v)
```

This book's CodeSamples API closely follows that of the Linux kernel.

4.3.6 Per-CPU Variables

The Linux kernel uses `DEFINE_PER_CPU()` to define a per-CPU variable, `this_cpu_ptr()` to form a reference to this CPU's instance of a given per-CPU variable, `per_cpu()` to access a specified CPU's instance of a given per-CPU variable, along with many other special-purpose per-CPU operations.

Listing 4.30 shows this book's per-thread-variable API, which is patterned after the Linux kernel's per-CPU-variable API. This API provides the per-thread equivalent of global variables. Although this API is, strictly speaking, not necessary[14], it can provide a good userspace analogy to Linux kernel code.

Quick Quiz 4.33: How could you work around the lack of a per-thread-variable API on systems that do not provide it? ■

4.3.6.1 `DEFINE_PER_THREAD()`

The `DEFINE_PER_THREAD()` primitive defines a per-thread variable. Unfortunately, it is not possible to provide an initializer in the way permitted by the Linux kernel's `DEFINE_PER_CPU()` primitive, but there is an `init_per_thread()` primitive that permits easy runtime initialization.

4.3.6.2 `DECLARE_PER_THREAD()`

The `DECLARE_PER_THREAD()` primitive is a declaration in the C sense, as opposed to a definition. Thus, a `DECLARE_PER_THREAD()` primitive may be used to access a per-thread variable defined in some other file.

4.3.6.3 `per_thread()`

The `per_thread()` primitive accesses the specified thread's variable.

[14] You could instead use `__thread` or `_Thread_local`.

4.3.6.4 `__get_thread_var()`

The `__get_thread_var()` primitive accesses the current thread's variable.

4.3.6.5 `init_per_thread()`

The `init_per_thread()` primitive sets all threads' instances of the specified variable to the specified value. The Linux kernel accomplishes this via normal C initialization, relying in clever use of linker scripts and code executed during the CPU-online process.

4.3.6.6 Usage Example

Suppose that we have a counter that is incremented very frequently but read out quite rarely. As will become clear in Section 5.2, it is helpful to implement such a counter using a per-thread variable. Such a variable can be defined as follows:

```
DEFINE_PER_THREAD(int, counter);
```

The counter must be initialized as follows:

```
init_per_thread(counter, 0);
```

A thread can increment its instance of this counter as follows:

```
p_counter = &__get_thread_var(counter);
WRITE_ONCE(*p_counter, *p_counter + 1);
```

The value of the counter is then the sum of its instances. A snapshot of the value of the counter can thus be collected as follows:

```
for_each_thread(t)
  sum += READ_ONCE(per_thread(counter, t));
```

Again, it is possible to gain a similar effect using other mechanisms, but per-thread variables combine convenience and high performance, as will be shown in more detail in Section 5.2.

4.4 The Right Tool for the Job: How to Choose?

> If you get stuck, change your tools; it may free your thinking.
>
> *Paul Arden, abbreviated*

As a rough rule of thumb, use the simplest tool that will get the job done. If you can, simply program sequentially. If that is insufficient, try using a shell script to mediate parallelism. If the resulting shell-script `fork()`/`exec()` overhead (about 480 microseconds for a minimal C program on an Intel Core Duo laptop) is too large, try using the C-language `fork()` and `wait()` primitives. If the overhead of these primitives (about 80 microseconds for a minimal child process) is still too large, then you might need to use the POSIX threading primitives, choosing the appropriate locking and/or atomic-operation primitives. If the overhead of the POSIX threading primitives (typically sub-microsecond) is too great, then the primitives introduced in Chapter 9 may be required. Of course, the actual overheads will depend not only on your hardware, but most critically on the manner in which you use the primitives. Furthermore, always remember that interprocess communication and message-passing can be good alternatives to shared-memory multithreaded execution, especially when your code makes good use of the design principles called out in Chapter 6.

Quick Quiz 4.34: Wouldn't the shell normally use `vfork()` rather than `fork()`? ■

Because concurrency was added to the C standard several decades after the C language was first used to build concurrent systems, there are a number of ways of concurrently accessing shared variables. All else being equal, the C11 standard operations described in Section 4.2.6 should be your first stop. If you need to access a given shared variable both with plain accesses and atomically, then the modern GCC atomics described in Section 4.2.7 might work well for you. If you are working on an old codebase that uses the classic GCC `__sync` API, then you should review Section 4.2.5 as well as the relevant GCC documentation. If you are working on the Linux kernel or similar codebase that combines use of the `volatile` keyword with inline assembly, or if you need dependencies to provide ordering, look at the material presented in Section 4.3.4 as well as that in Chapter 15.

www.dbooks.org

Whatever approach you take, please keep in mind that randomly hacking multi-threaded code is a spectacularly bad idea, especially given that shared-memory parallel systems use your own intelligence against you: The smarter you are, the deeper a hole you will dig for yourself before you realize that you are in trouble [Pok16]. Therefore, it is necessary to make the right design choices as well as the correct choice of individual primitives, as will be discussed at length in subsequent chapters.

Chapter 5

Counting

As easy as 1, 2, 3!

Unknown

Counting is perhaps the simplest and most natural thing a computer can do. However, counting efficiently and scalably on a large shared-memory multiprocessor can be quite challenging. Furthermore, the simplicity of the underlying concept of counting allows us to explore the fundamental issues of concurrency without the distractions of elaborate data structures or complex synchronization primitives. Counting therefore provides an excellent introduction to parallel programming.

This chapter covers a number of special cases for which there are simple, fast, and scalable counting algorithms. But first, let us find out how much you already know about concurrent counting.

Quick Quiz 5.1: Why should efficient and scalable counting be hard??? After all, computers have special hardware for the sole purpose of doing counting!!! ■

Quick Quiz 5.2: Network-packet counting problem. Suppose that you need to collect statistics on the number of networking packets transmitted and received. Packets might be transmitted or received by any CPU on the system. Suppose further that your system is capable of handling millions of packets per second per CPU, and that a systems-monitoring package reads the count every five seconds. How would you implement this counter? ■

Quick Quiz 5.3: Approximate structure-allocation limit problem. Suppose that you need to maintain a count of the number of structures allocated in order to fail any allocations once the number of structures in use exceeds a limit (say, 10,000). Suppose further that the structures are short-lived, the limit is rarely exceeded, and a "sloppy" approximate limit is acceptable. ■

Quick Quiz 5.4: Exact structure-allocation limit problem. Suppose that you need to maintain a count of the number of structures allocated in order to fail any allocations once the number of structures in use exceeds an exact limit (again, say 10,000). Suppose further that these structures are short-lived, and that the limit is rarely exceeded, that there is almost always at least one structure in use, and suppose further still that it is necessary to know exactly when this counter reaches zero, for example, in order to free up some memory that is not required unless there is at least one structure in use. ■

Quick Quiz 5.5: Removable I/O device access-count problem. Suppose that you need to maintain a reference count on a heavily used removable mass-storage device, so that you can tell the user when it is safe to remove the device. As usual, the user indicates a desire to remove the device, and the system tells the user when it is safe to do so. ■

Section 5.1 shows why counting is non-trivial. Sections 5.2 and 5.3 investigate network-packet counting and approximate structure-allocation limits, respectively. Section 5.4 takes on exact structure-allocation limits. Finally, Section 5.5 presents performance measurements and discussion.

Sections 5.1 and 5.2 contain introductory material, while the remaining sections are more advanced.

5.1 Why Isn't Concurrent Counting Trivial?

Seek simplicity, and distrust it.

Alfred North Whitehead

Let's start with something simple, for example, the straightforward use of arithmetic shown in Listing 5.1 (`count_nonatomic.c`). Here, we have a counter on line 1, we increment it on line 5, and we read out its value on line 10. What could be simpler?

Edition.2-rc11

www.dbooks.org

Listing 5.1: Just Count!

```
1 unsigned long counter = 0;
2
3 static __inline__ void inc_count(void)
4 {
5   WRITE_ONCE(counter, READ_ONCE(counter) + 1);
6 }
7
8 static __inline__ unsigned long read_count(void)
9 {
10  return READ_ONCE(counter);
11 }
```

Listing 5.2: Just Count Atomically!

```
1 atomic_t counter = ATOMIC_INIT(0);
2
3 static __inline__ void inc_count(void)
4 {
5   atomic_inc(&counter);
6 }
7
8 static __inline__ long read_count(void)
9 {
10  return atomic_read(&counter);
11 }
```

Quick Quiz 5.6: One thing that could be simpler is ++ instead of that concatenation of `READ_ONCE()` and `WRITE_ONCE()`. Why all that extra typing??? ■

This approach has the additional advantage of being blazingly fast if you are doing lots of reading and almost no incrementing, and on small systems, the performance is excellent.

There is just one large fly in the ointment: this approach can lose counts. On my six-core x86 laptop, a short run invoked `inc_count()` 285,824,000 times, but the final value of the counter was only 35,385,525. Although approximation does have a large place in computing, loss of 87 % of the counts is a bit excessive.

Quick Quiz 5.7: But can't a smart compiler prove that line 5 of Listing 5.1 is equivalent to the ++ operator and produce an x86 add-to-memory instruction? And won't the CPU cache cause this to be atomic? ■

Quick Quiz 5.8: The 8-figure accuracy on the number of failures indicates that you really did test this. Why would it be necessary to test such a trivial program, especially when the bug is easily seen by inspection? ■

The straightforward way to count accurately is to use atomic operations, as shown in Listing 5.2 (`count_atomic.c`). Line 1 defines an atomic variable, line 5 atomically increments it, and line 10 reads it out. Because this is atomic, it keeps perfect count. However, it is slower: on my six-core x86 laptop, it is more than twenty

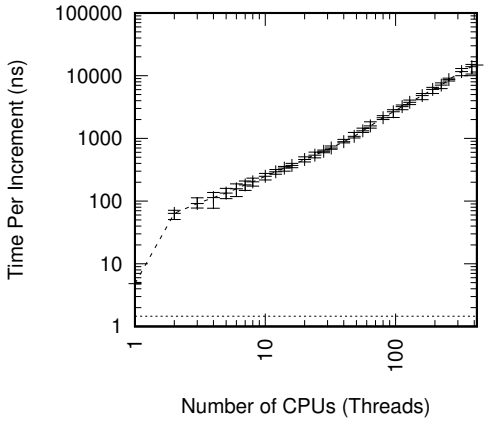


Figure 5.1: Atomic Increment Scalability on x86

times slower than non-atomic increment, even when only a single thread is incrementing.[1]

This poor performance should not be a surprise, given the discussion in Chapter 3, nor should it be a surprise that the performance of atomic increment gets slower as the number of CPUs and threads increase, as shown in Figure 5.1. In this figure, the horizontal dashed line resting on the x axis is the ideal performance that would be achieved by a perfectly scalable algorithm: with such an algorithm, a given increment would incur the same overhead that it would in a single-threaded program. Atomic increment of a single global variable is clearly decidedly non-ideal, and gets multiple orders of magnitude worse with additional CPUs.

Quick Quiz 5.9: Why doesn't the horizontal dashed line on the x axis meet the diagonal line at $x = 1$? ■

Quick Quiz 5.10: But atomic increment is still pretty fast. And incrementing a single variable in a tight loop sounds pretty unrealistic to me, after all, most of the program's execution should be devoted to actually doing work, not accounting for the work it has done! Why should I care about making this go faster? ■

For another perspective on global atomic increment, consider Figure 5.2. In order for each CPU to get a chance to increment a given global variable, the cache line containing that variable must circulate among all

[1] Interestingly enough, non-atomically incrementing a counter will advance the counter more quickly than atomically incrementing the counter. Of course, if your only goal is to make the counter increase quickly, an easier approach is to simply assign a large value to the counter. Nevertheless, there is likely to be a role for algorithms that use carefully relaxed notions of correctness in order to gain greater performance and scalability [And91, ACMS03, Rin13, Ung11].

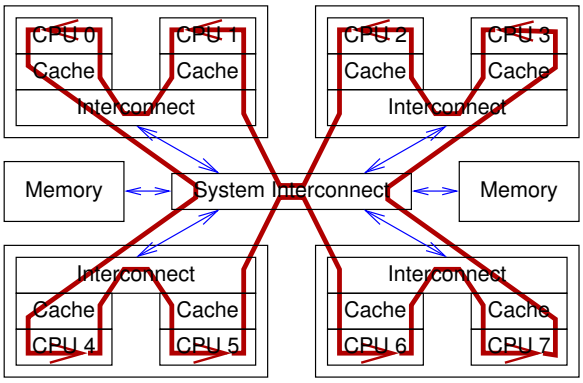


Figure 5.2: Data Flow For Global Atomic Increment

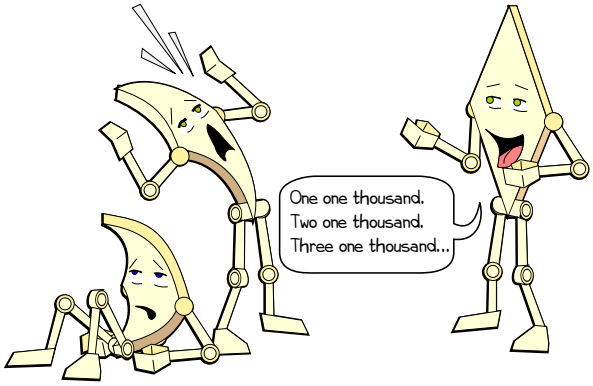


Figure 5.3: Waiting to Count

the CPUs, as shown by the red arrows. Such circulation will take significant time, resulting in the poor performance seen in Figure 5.1, which might be thought of as shown in Figure 5.3. The following sections discuss high-performance counting, which avoids the delays inherent in such circulation.

Quick Quiz 5.11: But why can't CPU designers simply ship the addition operation to the data, avoiding the need to circulate the cache line containing the global variable being incremented? ■

Listing 5.3: Array-Based Per-Thread Statistical Counters

```
1  DEFINE_PER_THREAD(unsigned long, counter);
2
3  static __inline__ void inc_count(void)
4  {
5    unsigned long *p_counter = &__get_thread_var(counter);
6
7    WRITE_ONCE(*p_counter, *p_counter + 1);
8  }
9
10 static __inline__ unsigned long read_count(void)
11 {
12   int t;
13   unsigned long sum = 0;
14
15   for_each_thread(t)
16     sum += READ_ONCE(per_thread(counter, t));
17   return sum;
18 }
```

5.2 Statistical Counters

> Facts are stubborn things, but statistics are pliable.
>
> *Mark Twain*

This section covers the common special case of statistical counters, where the count is updated extremely frequently and the value is read out rarely, if ever. These will be used to solve the network-packet counting problem posed in Quick Quiz 5.2.

5.2.1 Design

Statistical counting is typically handled by providing a counter per thread (or CPU, when running in the kernel), so that each thread updates its own counter, as was foreshadowed in Section 4.3.6 on page 46. The aggregate value of the counters is read out by simply summing up all of the threads' counters, relying on the commutative and associative properties of addition. This is an example of the Data Ownership pattern that will be introduced in Section 6.3.4 on page 85.

Quick Quiz 5.12: But doesn't the fact that C's "integers" are limited in size complicate things? ■

5.2.2 Array-Based Implementation

One way to provide per-thread variables is to allocate an array with one element per thread (presumably cache aligned and padded to avoid false sharing).

Quick Quiz 5.13: An array??? But doesn't that limit the number of threads? ■

www.dbooks.org

Such an array can be wrapped into per-thread primitives, as shown in Listing 5.3 (`count_stat.c`). Line 1 defines an array containing a set of per-thread counters of type `unsigned long` named, creatively enough, `counter`.

Lines 3–8 show a function that increments the counters, using the `__get_thread_var()` primitive to locate the currently running thread's element of the `counter` array. Because this element is modified only by the corresponding thread, non-atomic increment suffices. However, this code uses `WRITE_ONCE()` to prevent destructive compiler optimizations. For but one example, the compiler is within its rights to use a to-be-stored-to location as temporary storage, thus writing what would be for all intents and purposes garbage to that location just before doing the desired store. This could of course be rather confusing to anything attempting to read out the count. The use of `WRITE_ONCE()` prevents this optimization and others besides.

Quick Quiz 5.14: What other nasty optimizations could GCC apply? ■

Lines 10–18 show a function that reads out the aggregate value of the counter, using the `for_each_thread()` primitive to iterate over the list of currently running threads, and using the `per_thread()` primitive to fetch the specified thread's counter. This code also uses `READ_ONCE()` to ensure that the compiler doesn't optimize these loads into oblivion. For but one example, a pair of consecutive calls to `read_count()` might be inlined, and an intrepid optimizer might notice that the same locations were being summed and thus incorrectly conclude that it would be simply wonderful to sum them once and use the resulting value twice. This sort of optimization might be rather frustrating to people expecting later `read_count()` calls to account for the activities of other threads. The use of `READ_ONCE()` prevents this optimization and others besides.

Quick Quiz 5.15: How does the per-thread `counter` variable in Listing 5.3 get initialized? ■

Quick Quiz 5.16: How is the code in Listing 5.3 supposed to permit more than one counter? ■

This approach scales linearly with increasing number of updater threads invoking `inc_count()`. As is shown by the green arrows on each CPU in Figure 5.4, the reason for this is that each CPU can make rapid progress incrementing its thread's variable, without any expensive cross-system communication. As such, this section solves the network-packet counting problem presented at the beginning of this chapter.

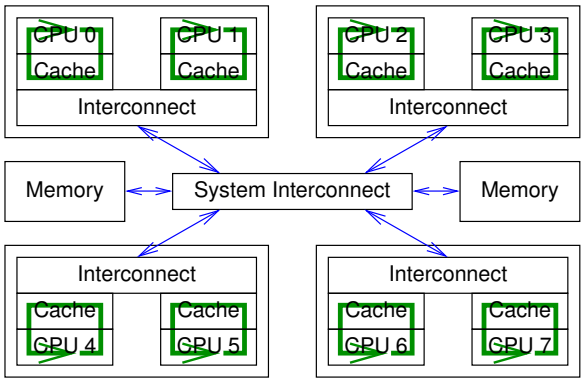


Figure 5.4: Data Flow For Per-Thread Increment

Quick Quiz 5.17: The read operation takes time to sum up the per-thread values, and during that time, the counter could well be changing. This means that the value returned by `read_count()` in Listing 5.3 will not necessarily be exact. Assume that the counter is being incremented at rate r counts per unit time, and that `read_count()`'s execution consumes Δ units of time. What is the expected error in the return value? ■

However, many implementations provide cheaper mechanisms for per-thread data that are free from arbitrary array-size limits. This is the topic of the next section.

5.2.3 Per-Thread-Variable-Based Implementation

GCC provides an `__thread` storage class that provides per-thread storage. This can be used as shown in Listing 5.4 (`count_end.c`) to implement a statistical counter that not only scales well and avoids arbitrary thread-number limits, but that also incurs little or no performance penalty to incrementers compared to simple non-atomic increment.

Lines 1–4 define needed variables: `counter` is the per-thread counter variable, the `counterp[]` array allows threads to access each others' counters, `finalcount` accumulates the total as individual threads exit, and `final_mutex` coordinates between threads accumulating the total value of the counter and exiting threads.

Quick Quiz 5.18: Doesn't that explicit `counterp` array in Listing 5.4 reimpose an arbitrary limit on the number of threads? Why doesn't GCC provide a `per_thread()` interface, similar to the Linux kernel's `per_cpu()` primitive,

Listing 5.4: Per-Thread Statistical Counters

```
1  unsigned long __thread counter = 0;
2  unsigned long *counterp[NR_THREADS] = { NULL };
3  unsigned long finalcount = 0;
4  DEFINE_SPINLOCK(final_mutex);
5
6  static __inline__ void inc_count(void)
7  {
8    WRITE_ONCE(counter, counter + 1);
9  }
10
11 static __inline__ unsigned long read_count(void)
12 {
13   int t;
14   unsigned long sum;
15
16   spin_lock(&final_mutex);
17   sum = finalcount;
18   for_each_thread(t)
19     if (counterp[t] != NULL)
20       sum += READ_ONCE(*counterp[t]);
21   spin_unlock(&final_mutex);
22   return sum;
23 }
24
25 void count_register_thread(unsigned long *p)
26 {
27   int idx = smp_thread_id();
28
29   spin_lock(&final_mutex);
30   counterp[idx] = &counter;
31   spin_unlock(&final_mutex);
32 }
33
34 void count_unregister_thread(int nthreadsexpected)
35 {
36   int idx = smp_thread_id();
37
38   spin_lock(&final_mutex);
39   finalcount += counter;
40   counterp[idx] = NULL;
41   spin_unlock(&final_mutex);
42 }
```

to allow threads to more easily access each others' per-thread variables? ■

The `inc_count()` function used by updaters is quite simple, as can be seen on lines 6–9.

The `read_count()` function used by readers is a bit more complex. Line 16 acquires a lock to exclude exiting threads, and line 21 releases it. Line 17 initializes the sum to the count accumulated by those threads that have already exited, and lines 18–20 sum the counts being accumulated by threads currently running. Finally, line 22 returns the sum.

Quick Quiz 5.19: Doesn't the check for `NULL` on line 19 of Listing 5.4 add extra branch mispredictions? Why not have a variable set permanently to zero, and point unused counter-pointers to that variable rather than setting them to `NULL`? ■

Quick Quiz 5.20: Why on earth do we need something as heavyweight as a *lock* guarding the summation in the function `read_count()` in Listing 5.4? ■

Lines 25–32 show the `count_register_thread()` function, which must be called by each thread before its first use of this counter. This function simply sets up this thread's element of the `counterp[]` array to point to its per-thread `counter` variable.

Quick Quiz 5.21: Why on earth do we need to acquire the lock in `count_register_thread()` in Listing 5.4? It is a single properly aligned machine-word store to a location that no other thread is modifying, so it should be atomic anyway, right? ■

Lines 34–42 show the `count_unregister_thread()` function, which must be called prior to exit by each thread that previously called `count_register_thread()`. Line 38 acquires the lock, and line 41 releases it, thus excluding any calls to `read_count()` as well as other calls to `count_unregister_thread()`. Line 39 adds this thread's `counter` to the global `finalcount`, and then line 40 `NULL`s out its `counterp[]` array entry. A subsequent call to `read_count()` will see the exiting thread's count in the global `finalcount`, and will skip the exiting thread when sequencing through the `counterp[]` array, thus obtaining the correct total.

This approach gives updaters almost exactly the same performance as a non-atomic add, and also scales linearly. On the other hand, concurrent reads contend for a single global lock, and therefore perform poorly and scale abysmally. However, this is not a problem for statistical counters, where incrementing happens often and readout happens almost never. Of course, this approach is considerably more complex than the array-based scheme, due to the fact that a given thread's per-thread variables vanish when that thread exits.

Quick Quiz 5.22: Fine, but the Linux kernel doesn't have to acquire a lock when reading out the aggregate value of per-CPU counters. So why should user-space code need to do this??? ■

Both the array-based and `__thread`-based approaches offer excellent update-side performance and scalability. However, these benefits result in large read-side expense for large numbers of threads. The next section shows one way to reduce read-side expense while still retaining the update-side scalability.

www.dbooks.org

5.2.4 Eventually Consistent Implementation

One way to retain update-side scalability while greatly improving read-side performance is to weaken consistency requirements. The counting algorithm in the previous section is guaranteed to return a value between the value that an ideal counter would have taken on near the beginning of `read_count()`'s execution and that near the end of `read_count()`'s execution. *Eventual consistency* [Vog09] provides a weaker guarantee: in absence of calls to `inc_count()`, calls to `read_count()` will eventually return an accurate count.

We exploit eventual consistency by maintaining a global counter. However, updaters only manipulate their per-thread counters. A separate thread is provided to transfer counts from the per-thread counters to the global counter. Readers simply access the value of the global counter. If updaters are active, the value used by the readers will be out of date, however, once updates cease, the global counter will eventually converge on the true value—hence this approach qualifies as eventually consistent.

The implementation is shown in Listing 5.5 (`count_stat_eventual.c`). Lines 1–2 show the per-thread variable and the global variable that track the counter's value, and line 3 shows `stopflag` which is used to coordinate termination (for the case where we want to terminate the program with an accurate counter value). The `inc_count()` function shown on lines 5–10 is similar to its counterpart in Listing 5.3. The `read_count()` function shown on lines 12–15 simply returns the value of the `global_count` variable.

However, the `count_init()` function on lines 36–46 creates the `eventual()` thread shown on lines 17–34, which cycles through all the threads, summing the per-thread local `counter` and storing the sum to the `global_count` variable. The `eventual()` thread waits an arbitrarily chosen one millisecond between passes.

The `count_cleanup()` function on lines 48–54 coordinates termination. The calls to `smp_mb()` here and in `eventual()` ensure that all updates to `global_count` are visible to code following the call to `count_cleanup()`.

This approach gives extremely fast counter read-out while still supporting linear counter-update scalability. However, this excellent read-side performance and update-side scalability comes at the cost of the additional thread running `eventual()`.

Listing 5.5: Array-Based Per-Thread Eventually Consistent Counters

```
1  DEFINE_PER_THREAD(unsigned long, counter);
2  unsigned long global_count;
3  int stopflag;
4
5  static __inline__ void inc_count(void)
6  {
7    unsigned long *p_counter = &__get_thread_var(counter);
8
9    WRITE_ONCE(*p_counter, *p_counter + 1);
10 }
11
12 static __inline__ unsigned long read_count(void)
13 {
14   return READ_ONCE(global_count);
15 }
16
17 void *eventual(void *arg)
18 {
19   int t;
20   unsigned long sum;
21
22   while (READ_ONCE(stopflag) < 3) {
23     sum = 0;
24     for_each_thread(t)
25       sum += READ_ONCE(per_thread(counter, t));
26     WRITE_ONCE(global_count, sum);
27     poll(NULL, 0, 1);
28     if (READ_ONCE(stopflag)) {
29       smp_mb();
30       WRITE_ONCE(stopflag, stopflag + 1);
31     }
32   }
33   return NULL;
34 }
35
36 void count_init(void)
37 {
38   int en;
39   thread_id_t tid;
40
41   en = pthread_create(&tid, NULL, eventual, NULL);
42   if (en != 0) {
43     fprintf(stderr, "pthread_create: %s\n", strerror(en));
44     exit(EXIT_FAILURE);
45   }
46 }
47
48 void count_cleanup(void)
49 {
50   WRITE_ONCE(stopflag, 1);
51   while (READ_ONCE(stopflag) < 3)
52     poll(NULL, 0, 1);
53   smp_mb();
54 }
```

Quick Quiz 5.23: Why doesn't `inc_count()` in Listing 5.5 need to use atomic instructions? After all, we now have multiple threads accessing the per-thread counters! ■

Quick Quiz 5.24: Won't the single global thread in the function `eventual()` of Listing 5.5 be just as severe a bottleneck as a global lock would be? ■

Quick Quiz 5.25: Won't the estimate returned by `read_count()` in Listing 5.5 become increasingly inaccurate as the number of threads rises? ■

Quick Quiz 5.26: Given that in the eventually-consistent algorithm shown in Listing 5.5 both reads and updates have extremely low overhead and are extremely scalable, why would anyone bother with the implementation described in Section 5.2.2, given its costly read-side code? ■

Quick Quiz 5.27: What is the accuracy of the estimate returned by `read_count()` in Listing 5.5? ■

5.2.5 Discussion

These three implementations show that it is possible to obtain near-uniprocessor performance for statistical counters, despite running on a parallel machine.

Quick Quiz 5.28: What fundamental difference is there between counting packets and counting the total number of bytes in the packets, given that the packets vary in size? ■

Quick Quiz 5.29: Given that the reader must sum all the threads' counters, this counter-read operation could take a long time given large numbers of threads. Is there any way that the increment operation can remain fast and scalable while allowing readers to also enjoy not only reasonable performance and scalability, but also good accuracy? ■

Given what has been presented in this section, you should now be able to answer the Quick Quiz about statistical counters for networking near the beginning of this chapter.

5.3 Approximate Limit Counters

> An approximate answer to the right problem is worth a good deal more than an exact answer to an approximate problem.
>
> *John Tukey*

Another special case of counting involves limit-checking. For example, as noted in the approximate structure-allocation limit problem in Quick Quiz 5.3, suppose that you need to maintain a count of the number of structures allocated in order to fail any allocations once the number of structures in use exceeds a limit, in this case, 10,000. Suppose further that these structures are short-lived, that this limit is rarely exceeded, and that this limit is approximate in that it is OK to exceed it sometimes by some bounded amount (see Section 5.4 if you instead need the limit to be exact).

5.3.1 Design

One possible design for limit counters is to divide the limit of 10,000 by the number of threads, and give each thread a fixed pool of structures. For example, given 100 threads, each thread would manage its own pool of 100 structures. This approach is simple, and in some cases works well, but it does not handle the common case where a given structure is allocated by one thread and freed by another [MS93]. On the one hand, if a given thread takes credit for any structures it frees, then the thread doing most of the allocating runs out of structures, while the threads doing most of the freeing have lots of credits that they cannot use. On the other hand, if freed structures are credited to the CPU that allocated them, it will be necessary for CPUs to manipulate each others' counters, which will require expensive atomic instructions or other means of communicating between threads.[2]

In short, for many important workloads, we cannot fully partition the counter. Given that partitioning the counters was what brought the excellent update-side performance for the three schemes discussed in Section 5.2, this might be grounds for some pessimism. However, the eventually consistent algorithm presented in Section 5.2.4 provides an interesting hint. Recall that this algorithm kept two sets of books, a per-thread `counter` variable for updaters and a `global_count` variable for readers, with an `eventual()`

[2] That said, if each structure will always be freed by the same CPU (or thread) that allocated it, then this simple partitioning approach works extremely well.

www.dbooks.org

thread that periodically updated global_count to be eventually consistent with the values of the per-thread counter. The per-thread counter perfectly partitioned the counter value, while global_count kept the full value.

For limit counters, we can use a variation on this theme where we *partially partition* the counter. For example, consider four threads with each having not only a per-thread counter, but also a per-thread maximum value (call it countermax).

But then what happens if a given thread needs to increment its counter, but counter is equal to its countermax? The trick here is to move half of that thread's counter value to a globalcount, then increment counter. For example, if a given thread's counter and countermax variables were both equal to 10, we do the following:

1. Acquire a global lock.
2. Add five to globalcount.
3. To balance out the addition, subtract five from this thread's counter.
4. Release the global lock.
5. Increment this thread's counter, resulting in a value of six.

Although this procedure still requires a global lock, that lock need only be acquired once for every five increment operations, greatly reducing that lock's level of contention. We can reduce this contention as low as we wish by increasing the value of countermax. However, the corresponding penalty for increasing the value of countermax is reduced accuracy of globalcount. To see this, note that on a four-CPU system, if countermax is equal to ten, globalcount will be in error by at most 40 counts. In contrast, if countermax is increased to 100, globalcount might be in error by as much as 400 counts.

This raises the question of just how much we care about globalcount's deviation from the aggregate value of the counter, where this aggregate value is the sum of globalcount and each thread's counter variable. The answer to this question depends on how far the aggregate value is from the counter's limit (call it globalcountmax). The larger the difference between these two values, the larger countermax can be without risk of exceeding the globalcountmax limit. This means that the value of a given thread's countermax variable can be set based on this difference. When far from the limit, the countermax per-thread variables are set to large values to optimize for performance and scalability, while when close to the limit, these same variables are set to small values to minimize the error in the checks against the globalcountmax limit.

This design is an example of *parallel fastpath*, which is an important design pattern in which the common case executes with no expensive instructions and no interactions between threads, but where occasional use is also made of a more conservatively designed (and higher overhead) global algorithm. This design pattern is covered in more detail in Section 6.4.

Listing 5.6: Simple Limit Counter Variables

```
1 unsigned long __thread counter = 0;
2 unsigned long __thread countermax = 0;
3 unsigned long globalcountmax = 10000;
4 unsigned long globalcount = 0;
5 unsigned long globalreserve = 0;
6 unsigned long *counterp[NR_THREADS] = { NULL };
7 DEFINE_SPINLOCK(gblcnt_mutex);
```

5.3.2 Simple Limit Counter Implementation

Listing 5.6 shows both the per-thread and global variables used by this implementation. The per-thread counter and countermax variables are the corresponding thread's local counter and the upper bound on that counter, respectively. The globalcountmax variable on line 3 contains the upper bound for the aggregate counter, and the globalcount variable on line 4 is the global counter. The sum of globalcount and each thread's counter gives the aggregate value of the overall counter. The globalreserve variable on line 5 is at least the sum of all of the per-thread countermax variables. The relationship among these variables is shown by Figure 5.5:

1. The sum of globalcount and globalreserve must be less than or equal to globalcountmax.
2. The sum of all threads' countermax values must be less than or equal to globalreserve.
3. Each thread's counter must be less than or equal to that thread's countermax.

Each element of the counterp[] array references the corresponding thread's counter variable, and, finally, the gblcnt_mutex spinlock guards all of the global variables, in other words, no thread is permitted to access or modify

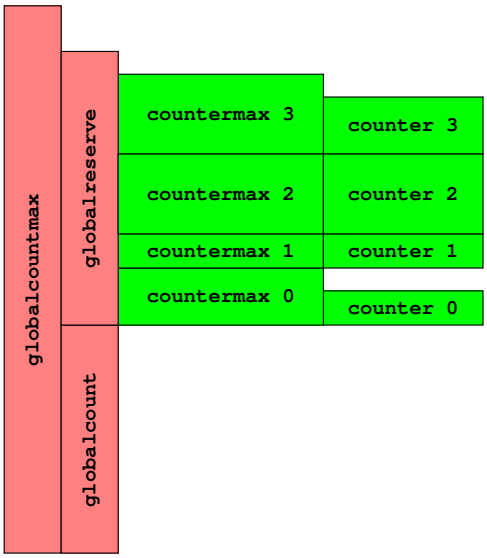


Figure 5.5: Simple Limit Counter Variable Relationships

any of the global variables unless it has acquired `gblcnt_mutex`.

Listing 5.7 shows the `add_count()`, `sub_count()`, and `read_count()` functions (`count_lim.c`).

Quick Quiz 5.30: Why does Listing 5.7 provide `add_count()` and `sub_count()` instead of the `inc_count()` and `dec_count()` interfaces show in Section 5.2? ■

Lines 1–18 show `add_count()`, which adds the specified value `delta` to the counter. Line 3 checks to see if there is room for `delta` on this thread's `counter`, and, if so, line 4 adds it and line 5 returns success. This is the `add_counter()` fastpath, and it does no atomic operations, references only per-thread variables, and should not incur any cache misses.

Quick Quiz 5.31: What is with the strange form of the condition on line 3 of Listing 5.7? Why not the more intuitive form of the fastpath shown in Listing 5.8? ■

If the test on line 3 fails, we must access global variables, and thus must acquire `gblcnt_mutex` on line 7, which we release on line 11 in the failure case or on line 16 in the success case. Line 8 invokes `globalize_count()`, shown in Listing 5.9, which clears the thread-local variables, adjusting the global variables as needed, thus simplifying global processing. (But don't take *my* word for it, try coding it yourself!) Lines 9 and 10 check to see if addition of `delta` can be accommodated, with the meaning of the expression preceding the less-than sign shown in Fig-

Listing 5.7: Simple Limit Counter Add, Subtract, and Read

```
1  static __inline__ int add_count(unsigned long delta)
2  {
3    if (countermax - counter >= delta) {
4      WRITE_ONCE(counter, counter + delta);
5      return 1;
6    }
7    spin_lock(&gblcnt_mutex);
8    globalize_count();
9    if (globalcountmax -
10       globalcount - globalreserve < delta) {
11     spin_unlock(&gblcnt_mutex);
12     return 0;
13   }
14   globalcount += delta;
15   balance_count();
16   spin_unlock(&gblcnt_mutex);
17   return 1;
18 }
19
20 static __inline__ int sub_count(unsigned long delta)
21 {
22   if (counter >= delta) {
23     WRITE_ONCE(counter, counter - delta);
24     return 1;
25   }
26   spin_lock(&gblcnt_mutex);
27   globalize_count();
28   if (globalcount < delta) {
29     spin_unlock(&gblcnt_mutex);
30     return 0;
31   }
32   globalcount -= delta;
33   balance_count();
34   spin_unlock(&gblcnt_mutex);
35   return 1;
36 }
37
38 static __inline__ unsigned long read_count(void)
39 {
40   int t;
41   unsigned long sum;
42
43   spin_lock(&gblcnt_mutex);
44   sum = globalcount;
45   for_each_thread(t)
46     if (counterp[t] != NULL)
47       sum += READ_ONCE(*counterp[t]);
48   spin_unlock(&gblcnt_mutex);
49   return sum;
50 }
```

Listing 5.8: Intuitive Fastpath

```
3    if (counter + delta <= countermax) {
4      WRITE_ONCE(counter, counter + delta);
5      return 1;
6    }
```


www.dbooks.org

ure 5.5 as the difference in height of the two red (leftmost) bars. If the addition of `delta` cannot be accommodated, then line 11 (as noted earlier) releases `gblcnt_mutex` and line 12 returns indicating failure.

Otherwise, we take the slowpath. Line 14 adds `delta` to `globalcount`, and then line 15 invokes `balance_count()` (shown in Listing 5.9) in order to update both the global and the per-thread variables. This call to `balance_count()` will usually set this thread's `countermax` to re-enable the fastpath. Line 16 then releases `gblcnt_mutex` (again, as noted earlier), and, finally, line 17 returns indicating success.

Quick Quiz 5.32: Why does `globalize_count()` zero the per-thread variables, only to later call `balance_count()` to refill them in Listing 5.7? Why not just leave the per-thread variables non-zero? ■

Lines 20–36 show `sub_count()`, which subtracts the specified `delta` from the counter. Line 22 checks to see if the per-thread counter can accommodate this subtraction, and, if so, line 23 does the subtraction and line 24 returns success. These lines form `sub_count()`'s fastpath, and, as with `add_count()`, this fastpath executes no costly operations.

If the fastpath cannot accommodate subtraction of `delta`, execution proceeds to the slowpath on lines 26–35. Because the slowpath must access global state, line 26 acquires `gblcnt_mutex`, which is released either by line 29 (in case of failure) or by line 34 (in case of success). Line 27 invokes `globalize_count()`, shown in Listing 5.9, which again clears the thread-local variables, adjusting the global variables as needed. Line 28 checks to see if the counter can accommodate subtracting `delta`, and, if not, line 29 releases `gblcnt_mutex` (as noted earlier) and line 30 returns failure.

Quick Quiz 5.33: Given that `globalreserve` counted against us in `add_count()`, why doesn't it count for us in `sub_count()` in Listing 5.7? ■

Quick Quiz 5.34: Suppose that one thread invokes `add_count()` shown in Listing 5.7, and then another thread invokes `sub_count()`. Won't `sub_count()` return failure even though the value of the counter is non-zero? ■

If, on the other hand, line 28 finds that the counter *can* accommodate subtracting `delta`, we complete the slowpath. Line 32 does the subtraction and then line 33 invokes `balance_count()` (shown in Listing 5.9) in order to update both global and per-thread variables (hopefully re-enabling the fastpath). Then line 34 releases `gblcnt_mutex`, and line 35 returns success.

Listing 5.9: Simple Limit Counter Utility Functions

```
1  static __inline__ void globalize_count(void)
2  {
3    globalcount += counter;
4    counter = 0;
5    globalreserve -= countermax;
6    countermax = 0;
7  }
8
9  static __inline__ void balance_count(void)
10 {
11   countermax = globalcountmax -
12                globalcount - globalreserve;
13   countermax /= num_online_threads();
14   globalreserve += countermax;
15   counter = countermax / 2;
16   if (counter > globalcount)
17     counter = globalcount;
18   globalcount -= counter;
19 }
20
21 void count_register_thread(void)
22 {
23   int idx = smp_thread_id();
24
25   spin_lock(&gblcnt_mutex);
26   counterp[idx] = &counter;
27   spin_unlock(&gblcnt_mutex);
28 }
29
30 void count_unregister_thread(int nthreadsexpected)
31 {
32   int idx = smp_thread_id();
33
34   spin_lock(&gblcnt_mutex);
35   globalize_count();
36   counterp[idx] = NULL;
37   spin_unlock(&gblcnt_mutex);
38 }
```

Quick Quiz 5.35: Why have both `add_count()` and `sub_count()` in Listing 5.7? Why not simply pass a negative number to `add_count()`? ■

Lines 38–50 show `read_count()`, which returns the aggregate value of the counter. It acquires `gblcnt_mutex` on line 43 and releases it on line 48, excluding global operations from `add_count()` and `sub_count()`, and, as we will see, also excluding thread creation and exit. Line 44 initializes local variable `sum` to the value of `globalcount`, and then the loop spanning lines 45–47 sums the per-thread `counter` variables. Line 49 then returns the sum.

Listing 5.9 shows a number of utility functions used by the `add_count()`, `sub_count()`, and `read_count()` primitives shown in Listing 5.7.

Lines 1–7 show `globalize_count()`, which zeros the current thread's per-thread counters, adjusting the global variables appropriately. It is important to note that

this function does not change the aggregate value of the counter, but instead changes how the counter's current value is represented. Line 3 adds the thread's `counter` variable to `globalcount`, and line 4 zeroes `counter`. Similarly, line 5 subtracts the per-thread `countermax` from `globalreserve`, and line 6 zeroes `countermax`. It is helpful to refer to Figure 5.5 when reading both this function and `balance_count()`, which is next.

Lines 9–19 show `balance_count()`, which is roughly speaking the inverse of `globalize_count()`. This function's job is to set the current thread's `countermax` variable to the largest value that avoids the risk of the counter exceeding the `globalcountmax` limit. Changing the current thread's `countermax` variable of course requires corresponding adjustments to `counter`, `globalcount` and `globalreserve`, as can be seen by referring back to Figure 5.5. By doing this, `balance_count()` maximizes use of `add_count()`'s and `sub_count()`'s low-overhead fastpaths. As with `globalize_count()`, `balance_count()` is not permitted to change the aggregate value of the counter.

Lines 11–13 compute this thread's share of that portion of `globalcountmax` that is not already covered by either `globalcount` or `globalreserve`, and assign the computed quantity to this thread's `countermax`. Line 14 makes the corresponding adjustment to `globalreserve`. Line 15 sets this thread's `counter` to the middle of the range from zero to `countermax`. Line 16 checks to see whether `globalcount` can in fact accommodate this value of `counter`, and, if not, line 17 decreases `counter` accordingly. Finally, in either case, line 18 makes the corresponding adjustment to `globalcount`.

Quick Quiz 5.36: Why set `counter` to `countermax / 2` in line 15 of Listing 5.9? Wouldn't it be simpler to just take `countermax` counts? ■

It is helpful to look at a schematic depicting how the relationship of the counters changes with the execution of first `globalize_count()` and then `balance_count`, as shown in Figure 5.6. Time advances from left to right, with the leftmost configuration roughly that of Figure 5.5. The center configuration shows the relationship of these same counters after `globalize_count()` is executed by thread 0. As can be seen from the figure, thread 0's `counter` ("c 0" in the figure) is added to `globalcount`, while the value of `globalreserve` is reduced by this same amount. Both thread 0's `counter` and its `countermax` ("cm 0" in the figure) are reduced to zero. The other three threads' counters are unchanged. Note that this change did not affect the overall value of the counter, as indicated by the bottommost dotted line connecting the leftmost and center configurations. In other words, the sum of `globalcount` and the four threads' `counter` variables is the same in both configurations. Similarly, this change did not affect the sum of `globalcount` and `globalreserve`, as indicated by the upper dotted line.

The rightmost configuration shows the relationship of these counters after `balance_count()` is executed, again by thread 0. One-quarter of the remaining count, denoted by the vertical line extending up from all three configurations, is added to thread 0's `countermax` and half of that to thread 0's `counter`. The amount added to thread 0's `counter` is also subtracted from `globalcount` in order to avoid changing the overall value of the counter (which is again the sum of `globalcount` and the three threads' `counter` variables), again as indicated by the lowermost of the two dotted lines connecting the center and rightmost configurations. The `globalreserve` variable is also adjusted so that this variable remains equal to the sum of the four threads' `countermax` variables. Because thread 0's `counter` is less than its `countermax`, thread 0 can once again increment the counter locally.

Quick Quiz 5.37: In Figure 5.6, even though a quarter of the remaining count up to the limit is assigned to thread 0, only an eighth of the remaining count is consumed, as indicated by the uppermost dotted line connecting the center and the rightmost configurations. Why is that? ■

Lines 21–28 show `count_register_thread()`, which sets up state for newly created threads. This function simply installs a pointer to the newly created thread's `counter` variable into the corresponding entry of the `counterp[]` array under the protection of `gblcnt_mutex`.

Finally, lines 30–38 show `count_unregister_thread()`, which tears down state for a soon-to-be-exiting thread. Line 34 acquires `gblcnt_mutex` and line 37 releases it. Line 35 invokes `globalize_count()` to clear out this thread's counter state, and line 36 clears this thread's entry in the `counterp[]` array.

5.3.3 Simple Limit Counter Discussion

This type of counter is quite fast when aggregate values are near zero, with some overhead due to the comparison and branch in both `add_count()`'s and `sub_count()`'s fastpaths. However, the use of a per-thread `countermax` reserve means that `add_count()` can fail even when the aggregate value of the counter is nowhere near `globalcountmax`. Similarly, `sub_count()` can fail

www.dbooks.org

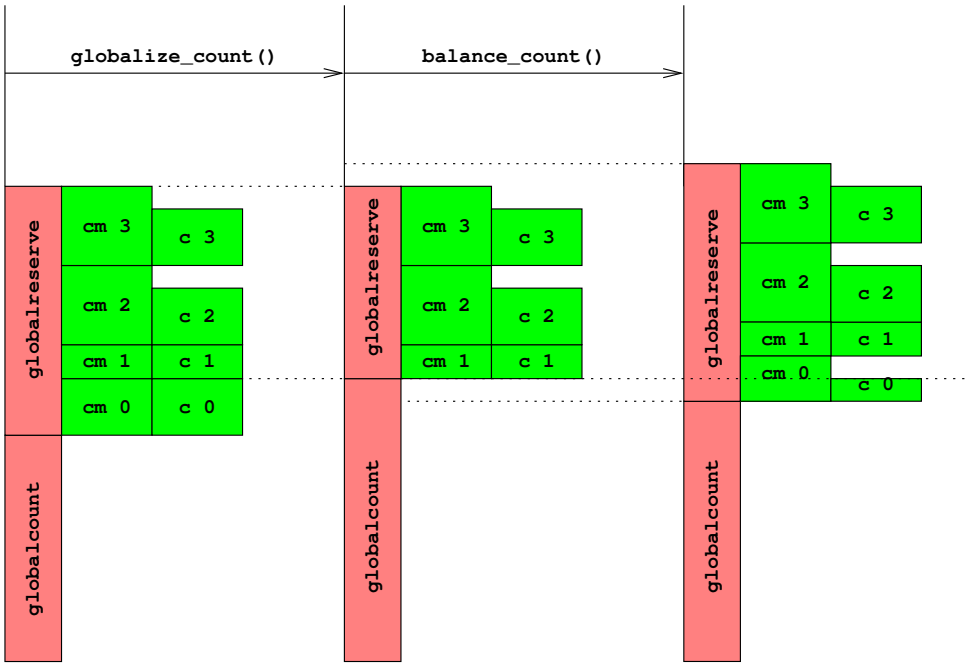


Figure 5.6: Schematic of Globalization and Balancing

even when the aggregate value of the counter is nowhere near zero.

In many cases, this is unacceptable. Even if the `globalcountmax` is intended to be an approximate limit, there is usually a limit to exactly how much approximation can be tolerated. One way to limit the degree of approximation is to impose an upper limit on the value of the per-thread `countermax` instances. This task is undertaken in the next section.

Listing 5.10: Approximate Limit Counter Variables

```
1 unsigned long __thread counter = 0;
2 unsigned long __thread countermax = 0;
3 unsigned long globalcountmax = 10000;
4 unsigned long globalcount = 0;
5 unsigned long globalreserve = 0;
6 unsigned long *counterp[NR_THREADS] = { NULL };
7 DEFINE_SPINLOCK(gblcnt_mutex);
8 #define MAX_COUNTERMAX 100
```

5.3.4 Approximate Limit Counter Implementation

Because this implementation (`count_lim_app.c`) is quite similar to that in the previous section (Listings 5.6, 5.7, and 5.9), only the changes are shown here. Listing 5.10 is identical to Listing 5.6, with the addition of `MAX_COUNTERMAX`, which sets the maximum permissible value of the per-thread `countermax` variable.

Similarly, Listing 5.11 is identical to the `balance_count()` function in Listing 5.9, with the addition of lines 6 and 7, which enforce the `MAX_COUNTERMAX` limit on the per-thread `countermax` variable.

Listing 5.11: Approximate Limit Counter Balancing

```
1 static void balance_count(void)
2 {
3   countermax = globalcountmax -
4                globalcount - globalreserve;
5   countermax /= num_online_threads();
6   if (countermax > MAX_COUNTERMAX)
7     countermax = MAX_COUNTERMAX;
8   globalreserve += countermax;
9   counter = countermax / 2;
10  if (counter > globalcount)
11    counter = globalcount;
12  globalcount -= counter;
13 }
```

5.3.5 Approximate Limit Counter Discussion

These changes greatly reduce the limit inaccuracy seen in the previous version, but present another problem: any given value of MAX_COUNTERMAX will cause a workload-dependent fraction of accesses to fall off the fastpath. As the number of threads increase, non-fastpath execution will become both a performance and a scalability problem. However, we will defer this problem and turn instead to counters with exact limits.

5.4 Exact Limit Counters

> Exactitude can be expensive. Spend wisely.
>
> *Unknown*

To solve the exact structure-allocation limit problem noted in Quick Quiz 5.4, we need a limit counter that can tell exactly when its limits are exceeded. One way of implementing such a limit counter is to cause threads that have reserved counts to give them up. One way to do this is to use atomic instructions. Of course, atomic instructions will slow down the fastpath, but on the other hand, it would be silly not to at least give them a try.

5.4.1 Atomic Limit Counter Implementation

Unfortunately, if one thread is to safely remove counts from another thread, both threads will need to atomically manipulate that thread's counter and countermax variables. The usual way to do this is to combine these two variables into a single variable, for example, given a 32-bit variable, using the high-order 16 bits to represent counter and the low-order 16 bits to represent countermax.

Quick Quiz 5.38: Why is it necessary to atomically manipulate the thread's counter and countermax variables as a unit? Wouldn't it be good enough to atomically manipulate them individually? ■

The variables and access functions for a simple atomic limit counter are shown in Listing 5.12 (count_lim_atomic.c). The counter and countermax variables in earlier algorithms are combined into the single variable counterandmax shown on line 1, with counter in the upper half and countermax in the lower half. This

Listing 5.12: Atomic Limit Counter Variables and Access Functions

```
1  atomic_t __thread counterandmax = ATOMIC_INIT(0);
2  unsigned long globalcountmax = 1 << 25;
3  unsigned long globalcount = 0;
4  unsigned long globalreserve = 0;
5  atomic_t *counterp[NR_THREADS] = { NULL };
6  DEFINE_SPINLOCK(gblcnt_mutex);
7  #define CM_BITS (sizeof(atomic_t) * 4)
8  #define MAX_COUNTERMAX ((1 << CM_BITS) - 1)
9
10 static __inline__ void
11 split_counterandmax_int(int cami, int *c, int *cm)
12 {
13   *c = (cami >> CM_BITS) & MAX_COUNTERMAX;
14   *cm = cami & MAX_COUNTERMAX;
15 }
16
17 static __inline__ void
18 split_counterandmax(atomic_t *cam, int *old, int *c, int *cm)
19 {
20   unsigned int cami = atomic_read(cam);
21
22   *old = cami;
23   split_counterandmax_int(cami, c, cm);
24 }
25
26 static __inline__ int merge_counterandmax(int c, int cm)
27 {
28   unsigned int cami;
29
30   cami = (c << CM_BITS) | cm;
31   return ((int)cami);
32 }
```

variable is of type atomic_t, which has an underlying representation of int.

Lines 2–6 show the definitions for globalcountmax, globalcount, globalreserve, counterp, and gblcnt_mutex, all of which take on roles similar to their counterparts in Listing 5.10. Line 7 defines CM_BITS, which gives the number of bits in each half of counterandmax, and line 8 defines MAX_COUNTERMAX, which gives the maximum value that may be held in either half of counterandmax.

Quick Quiz 5.39: In what way does line 7 of Listing 5.12 violate the C standard? ■

Lines 10–15 show the split_counterandmax_int() function, which, when given the underlying int from the atomic_t counterandmax variable, splits it into its counter (c) and countermax (cm) components. Line 13 isolates the most-significant half of this int, placing the result as specified by argument c, and line 14 isolates the least-significant half of this int, placing the result as specified by argument cm.

Lines 17–24 show the split_counterandmax() function, which picks up the underlying int from the specified variable on line 20, stores it as specified by the

www.dbooks.org

old argument on line 22, and then invokes split_counterandmax_int() to split it on line 23.

Quick Quiz 5.40: Given that there is only one counterandmax variable, why bother passing in a pointer to it on line 18 of Listing 5.12? ■

Lines 26–32 show the merge_counterandmax() function, which can be thought of as the inverse of split_counterandmax(). Line 30 merges the counter and countermax values passed in c and cm, respectively, and returns the result.

Quick Quiz 5.41: Why does merge_counterandmax() in Listing 5.12 return an int rather than storing directly into an atomic_t? ■

Listing 5.13 shows the add_count() and sub_count() functions.

Lines 1–32 show add_count(), whose fastpath spans lines 8–15, with the remainder of the function being the slowpath. Lines 8–14 of the fastpath form a compare-and-swap (CAS) loop, with the atomic_cmpxchg() primitives on lines 13–14 performing the actual CAS. Line 9 splits the current thread's counterandmax variable into its counter (in c) and countermax (in cm) components, while placing the underlying int into old. Line 10 checks whether the amount delta can be accommodated locally (taking care to avoid integer overflow), and if not, line 11 transfers to the slowpath. Otherwise, line 12 combines an updated counter value with the original countermax value into new. The atomic_cmpxchg() primitive on lines 13–14 then atomically compares this thread's counterandmax variable to old, updating its value to new if the comparison succeeds. If the comparison succeeds, line 15 returns success, otherwise, execution continues in the loop at line 8.

Quick Quiz 5.42: Yecch! Why the ugly goto on line 11 of Listing 5.13? Haven't you heard of the break statement??? ■

Quick Quiz 5.43: Why would the atomic_cmpxchg() primitive at lines 13–14 of Listing 5.13 ever fail? After all, we picked up its old value on line 9 and have not changed it! ■

Lines 16–31 of Listing 5.13 show add_count()'s slowpath, which is protected by gblcnt_mutex, which is acquired on line 17 and released on lines 24 and 30. Line 18 invokes globalize_count(), which moves this thread's state to the global counters. Lines 19–20 check whether the delta value can be accommodated by the current global state, and, if not, line 21 invokes flush_local_count() to flush all threads' local state to the

Listing 5.13: Atomic Limit Counter Add and Subtract

```
1  int add_count(unsigned long delta)
2  {
3    int c;
4    int cm;
5    int old;
6    int new;
7
8    do {
9      split_counterandmax(&counterandmax, &old, &c, &cm);
10     if (delta > MAX_COUNTERMAX || c + delta > cm)
11       goto slowpath;
12     new = merge_counterandmax(c + delta, cm);
13   } while (atomic_cmpxchg(&counterandmax,
14                           old, new) != old);
15   return 1;
16 slowpath:
17   spin_lock(&gblcnt_mutex);
18   globalize_count();
19   if (globalcountmax - globalcount -
20       globalreserve < delta) {
21     flush_local_count();
22     if (globalcountmax - globalcount -
23         globalreserve < delta) {
24       spin_unlock(&gblcnt_mutex);
25       return 0;
26     }
27   }
28   globalcount += delta;
29   balance_count();
30   spin_unlock(&gblcnt_mutex);
31   return 1;
32 }
33
34 int sub_count(unsigned long delta)
35 {
36   int c;
37   int cm;
38   int old;
39   int new;
40
41   do {
42     split_counterandmax(&counterandmax, &old, &c, &cm);
43     if (delta > c)
44       goto slowpath;
45     new = merge_counterandmax(c - delta, cm);
46   } while (atomic_cmpxchg(&counterandmax,
47                           old, new) != old);
48   return 1;
49  slowpath:
50   spin_lock(&gblcnt_mutex);
51   globalize_count();
52   if (globalcount < delta) {
53     flush_local_count();
54     if (globalcount < delta) {
55       spin_unlock(&gblcnt_mutex);
56       return 0;
57     }
58   }
59   globalcount -= delta;
60   balance_count();
61   spin_unlock(&gblcnt_mutex);
62   return 1;
63 }
```

Listing 5.14: Atomic Limit Counter Read

```
1 unsigned long read_count(void)
2 {
3   int c;
4   int cm;
5   int old;
6   int t;
7   unsigned long sum;
8
9   spin_lock(&gblcnt_mutex);
10  sum = globalcount;
11  for_each_thread(t)
12    if (counterp[t] != NULL) {
13      split_counterandmax(counterp[t], &old, &c, &cm);
14      sum += c;
15    }
16  spin_unlock(&gblcnt_mutex);
17  return sum;
18 }
```

Listing 5.15: Atomic Limit Counter Utility Functions 1

```
1 static void globalize_count(void)
2 {
3   int c;
4   int cm;
5   int old;
6
7   split_counterandmax(&counterandmax, &old, &c, &cm);
8   globalcount += c;
9   globalreserve -= cm;
10  old = merge_counterandmax(0, 0);
11  atomic_set(&counterandmax, old);
12 }
13
14 static void flush_local_count(void)
15 {
16  int c;
17  int cm;
18  int old;
19  int t;
20  int zero;
21
22  if (globalreserve == 0)
23    return;
24  zero = merge_counterandmax(0, 0);
25  for_each_thread(t)
26    if (counterp[t] != NULL) {
27      old = atomic_xchg(counterp[t], zero);
28      split_counterandmax_int(old, &c, &cm);
29      globalcount += c;
30      globalreserve -= cm;
31    }
32 }
```

global counters, and then lines 22–23 recheck whether `delta` can be accommodated. If, after all that, the addition of `delta` still cannot be accommodated, then line 24 releases `gblcnt_mutex` (as noted earlier), and then line 25 returns failure.

Otherwise, line 28 adds `delta` to the global counter, line 29 spreads counts to the local state if appropriate, line 30 releases `gblcnt_mutex` (again, as noted earlier), and finally, line 31 returns success.

Lines 34–63 of Listing 5.13 show `sub_count()`, which is structured similarly to `add_count()`, having a fastpath on lines 41–48 and a slowpath on lines 49–62. A line-by-line analysis of this function is left as an exercise to the reader.

Listing 5.14 shows `read_count()`. Line 9 acquires `gblcnt_mutex` and line 16 releases it. Line 10 initializes local variable `sum` to the value of `globalcount`, and the loop spanning lines 11–15 adds the per-thread counters to this sum, isolating each per-thread counter using `split_counterandmax` on line 13. Finally, line 17 returns the sum.

Listings 5.15 and 5.16 show the utility functions `globalize_count()`, `flush_local_count()`, `balance_count()`, `count_register_thread()`, and `count_unregister_thread()`. The code for `globalize_count()` is shown on lines 1–12, of Listing 5.15 and is similar to that of previous algorithms, with the addition of line 7, which is now required to split out `counter` and `countermax` from `counterandmax`.

The code for `flush_local_count()`, which moves all threads' local counter state to the global counter, is shown on lines 14–32. Line 22 checks to see if the value of `globalreserve` permits any per-thread counts, and, if not, line 23 returns. Otherwise, line 24 initializes local variable `zero` to a combined zeroed `counter` and `countermax`. The loop spanning lines 25–31 sequences through each thread. Line 26 checks to see if the current thread has counter state, and, if so, lines 27–30 move that state to the global counters. Line 27 atomically fetches the current thread's state while replacing it with zero. Line 28 splits this state into its `counter` (in local variable `c`) and `countermax` (in local variable `cm`) components. Line 29 adds this thread's `counter` to `globalcount`, while line 30 subtracts this thread's `countermax` from `globalreserve`.

Quick Quiz 5.44: What stops a thread from simply refilling its `counterandmax` variable immediately after `flush_local_count()` on line 14 of Listing 5.15 empties it? ■

Quick Quiz 5.45: What prevents concurrent execution of the fastpath of either `add_count()` or `sub_count()` from interfering with the `counterandmax` variable while `flush_local_count()` is accessing it on line 27 of Listing 5.15? ■

Lines 1–22 on Listing 5.16 show the code for `balance_count()`, which refills the calling thread's local `counterandmax` variable. This function is quite similar to that of the preceding algorithms, with changes required to handle the merged `counterandmax` variable. Detailed

www.dbooks.org

Listing 5.16: Atomic Limit Counter Utility Functions 2

```
1  static void balance_count(void)
2  {
3    int c;
4    int cm;
5    int old;
6    unsigned long limit;
7
8    limit = globalcountmax - globalcount -
9            globalreserve;
10   limit /= num_online_threads();
11   if (limit > MAX_COUNTERMAX)
12     cm = MAX_COUNTERMAX;
13   else
14     cm = limit;
15   globalreserve += cm;
16   c = cm / 2;
17   if (c > globalcount)
18     c = globalcount;
19   globalcount -= c;
20   old = merge_counterandmax(c, cm);
21   atomic_set(&counterandmax, old);
22 }
23
24 void count_register_thread(void)
25 {
26   int idx = smp_thread_id();
27
28   spin_lock(&gblcnt_mutex);
29   counterp[idx] = &counterandmax;
30   spin_unlock(&gblcnt_mutex);
31 }
32
33 void count_unregister_thread(int nthreadsexpected)
34 {
35   int idx = smp_thread_id();
36
37   spin_lock(&gblcnt_mutex);
38   globalize_count();
39   counterp[idx] = NULL;
40   spin_unlock(&gblcnt_mutex);
41 }
```

analysis of the code is left as an exercise for the reader, as it is with the `count_register_thread()` function starting on line 24 and the `count_unregister_thread()` function starting on line 33.

Quick Quiz 5.46: Given that the `atomic_set()` primitive does a simple store to the specified `atomic_t`, how can line 21 of `balance_count()` in Listing 5.16 work correctly in face of concurrent `flush_local_count()` updates to this variable? ■

The next section qualitatively evaluates this design.

5.4.2 Atomic Limit Counter Discussion

This is the first implementation that actually allows the counter to be run all the way to either of its limits, but it does so at the expense of adding atomic operations to the fastpaths, which slow down the fastpaths significantly on some systems. Although some workloads might tolerate this slowdown, it is worthwhile looking for algorithms with better read-side performance. One such algorithm uses a signal handler to steal counts from other threads. Because signal handlers run in the context of the signaled thread, atomic operations are not necessary, as shown in the next section.

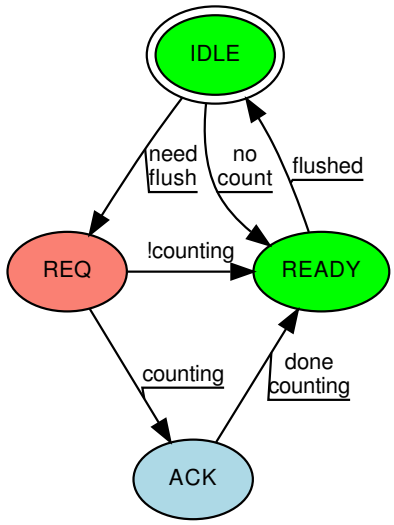


Figure 5.7: Signal-Theft State Machine

Quick Quiz 5.47: But signal handlers can be migrated to some other CPU while running. Doesn't this possibility require that atomic instructions and memory barriers are required to reliably communicate between a thread and a signal handler that interrupts that thread? ■

5.4.3 Signal-Theft Limit Counter Design

Even though per-thread state will now be manipulated only by the corresponding thread, there will still need to be synchronization with the signal handlers. This synchronization is provided by the state machine shown in Figure 5.7.

The state machine starts out in the IDLE state, and when `add_count()` or `sub_count()` find that the combination of the local thread's count and the global count cannot accommodate the request, the corresponding slowpath sets each thread's `theft` state to REQ (unless that thread has no count, in which case it transitions directly to READY). Only the slowpath, which holds the `gblcnt_mutex` lock, is permitted to transition from the IDLE state, as indicated by the green color.[3] The slowpath then sends a signal

[3] For those with black-and-white versions of this book, IDLE and READY are green, REQ is red, and ACK is blue.

to each thread, and the corresponding signal handler checks the corresponding thread's `theft` and `counting` variables. If the `theft` state is not REQ, then the signal handler is not permitted to change the state, and therefore simply returns. Otherwise, if the `counting` variable is set, indicating that the current thread's fastpath is in progress, the signal handler sets the `theft` state to ACK, otherwise to READY.

If the `theft` state is ACK, only the fastpath is permitted to change the `theft` state, as indicated by the blue color. When the fastpath completes, it sets the `theft` state to READY.

Once the slowpath sees a thread's `theft` state is READY, the slowpath is permitted to steal that thread's count. The slowpath then sets that thread's `theft` state to IDLE.

Quick Quiz 5.48: In Figure 5.7, why is the REQ `theft` state colored red? ■

Quick Quiz 5.49: In Figure 5.7, what is the point of having separate REQ and ACK `theft` states? Why not simplify the state machine by collapsing them into a single REQACK state? Then whichever of the signal handler or the fastpath gets there first could set the state to READY. ■

5.4.4 Signal-Theft Limit Counter Implementation

Listing 5.17 (`count_lim_sig.c`) shows the data structures used by the signal-theft based counter implementation. Lines 1–7 define the states and values for the per-thread theft state machine described in the preceding section. Lines 8–17 are similar to earlier implementations, with the addition of lines 14 and 15 to allow remote access to a thread's `countermax` and `theft` variables, respectively.

Listing 5.18 shows the functions responsible for migrating counts between per-thread variables and the global variables. Lines 1–7 show `globalize_count()`, which is identical to earlier implementations. Lines 9–19 show `flush_local_count_sig()`, which is the signal handler used in the theft process. Lines 11 and 12 check to see if the `theft` state is REQ, and, if not returns without change. Line 13 executes a memory barrier to ensure that the sampling of the theft variable happens before any change to that variable. Line 14 sets the `theft` state to ACK, and, if line 15 sees that this thread's fastpaths are not running, line 16 sets the `theft` state to READY.

Listing 5.17: Signal-Theft Limit Counter Data

```
1 #define THEFT_IDLE   0
2 #define THEFT_REQ    1
3 #define THEFT_ACK    2
4 #define THEFT_READY  3
5
6 int __thread theft = THEFT_IDLE;
7 int __thread counting = 0;
8 unsigned long __thread counter = 0;
9 unsigned long __thread countermax = 0;
10 unsigned long globalcountmax = 10000;
11 unsigned long globalcount = 0;
12 unsigned long globalreserve = 0;
13 unsigned long *counterp[NR_THREADS] = { NULL };
14 unsigned long *countermaxp[NR_THREADS] = { NULL };
15 int *theftp[NR_THREADS] = { NULL };
16 DEFINE_SPINLOCK(gblcnt_mutex);
17 #define MAX_COUNTERMAX 100
```

Quick Quiz 5.50: In Listing 5.18's function `flush_local_count_sig()`, why are there `READ_ONCE()` and `WRITE_ONCE()` wrappers around the uses of the `theft` per-thread variable? ■

Lines 21–49 show `flush_local_count()`, which is called from the slowpath to flush all threads' local counts. The loop spanning lines 26–34 advances the `theft` state for each thread that has local count, and also sends that thread a signal. Line 27 skips any non-existent threads. Otherwise, line 28 checks to see if the current thread holds any local count, and, if not, line 29 sets the thread's `theft` state to READY and line 30 skips to the next thread. Otherwise, line 32 sets the thread's `theft` state to REQ and line 33 sends the thread a signal.

Quick Quiz 5.51: In Listing 5.18, why is it safe for line 28 to directly access the other thread's `countermax` variable? ■

Quick Quiz 5.52: In Listing 5.18, why doesn't line 33 check for the current thread sending itself a signal? ■

Quick Quiz 5.53: The code shown in Listings 5.17 and 5.18 works with GCC and POSIX. What would be required to make it also conform to the ISO C standard? ■

The loop spanning lines 35–48 waits until each thread reaches READY state, then steals that thread's count. Lines 36–37 skip any non-existent threads, and the loop spanning lines 38–42 waits until the current thread's `theft` state becomes READY. Line 39 blocks for a millisecond to avoid priority-inversion problems, and if line 40 determines that the thread's signal has not yet arrived, line 41 resends the signal. Execution reaches line 43 when the thread's `theft` state becomes READY, so lines 43–46 do the thieving. Line 47 then sets the thread's `theft` state back to IDLE.

www.dbooks.org

Listing 5.18: Signal-Theft Limit Counter Value-Migration Functions

```
1  static void globalize_count(void)
2  {
3    globalcount += counter;
4    counter = 0;
5    globalreserve -= countermax;
6    countermax = 0;
7  }
8
9  static void flush_local_count_sig(int unused)
10 {
11   if (READ_ONCE(theft) != THEFT_REQ)
12     return;
13   smp_mb();
14   WRITE_ONCE(theft, THEFT_ACK);
15   if (!counting) {
16     WRITE_ONCE(theft, THEFT_READY);
17   }
18   smp_mb();
19 }
20
21 static void flush_local_count(void)
22 {
23   int t;
24   thread_id_t tid;
25
26   for_each_tid(t, tid)
27     if (theftp[t] != NULL) {
28       if (*countermaxp[t] == 0) {
29         WRITE_ONCE(*theftp[t], THEFT_READY);
30         continue;
31       }
32       WRITE_ONCE(*theftp[t], THEFT_REQ);
33       pthread_kill(tid, SIGUSR1);
34     }
35   for_each_tid(t, tid) {
36     if (theftp[t] == NULL)
37       continue;
38     while (READ_ONCE(*theftp[t]) != THEFT_READY) {
39       poll(NULL, 0, 1);
40       if (READ_ONCE(*theftp[t]) == THEFT_REQ)
41         pthread_kill(tid, SIGUSR1);
42     }
43     globalcount += *counterp[t];
44     *counterp[t] = 0;
45     globalreserve -= *countermaxp[t];
46     *countermaxp[t] = 0;
47     WRITE_ONCE(*theftp[t], THEFT_IDLE);
48   }
49 }
50
51 static void balance_count(void)
52 {
53   countermax = globalcountmax - globalcount -
54                globalreserve;
55   countermax /= num_online_threads();
56   if (countermax > MAX_COUNTERMAX)
57     countermax = MAX_COUNTERMAX;
58   globalreserve += countermax;
59   counter = countermax / 2;
60   if (counter > globalcount)
61     counter = globalcount;
62   globalcount -= counter;
63 }
```

Listing 5.19: Signal-Theft Limit Counter Add Function

```
1  int add_count(unsigned long delta)
2  {
3    int fastpath = 0;
4
5    WRITE_ONCE(counting, 1);
6    barrier();
7    if (READ_ONCE(theft) <= THEFT_REQ &&
8        countermax - counter >= delta) {
9      WRITE_ONCE(counter, counter + delta);
10     fastpath = 1;
11   }
12   barrier();
13   WRITE_ONCE(counting, 0);
14   barrier();
15   if (READ_ONCE(theft) == THEFT_ACK) {
16     smp_mb();
17     WRITE_ONCE(theft, THEFT_READY);
18   }
19   if (fastpath)
20     return 1;
21   spin_lock(&gblcnt_mutex);
22   globalize_count();
23   if (globalcountmax - globalcount -
24       globalreserve < delta) {
25     flush_local_count();
26     if (globalcountmax - globalcount -
27         globalreserve < delta) {
28       spin_unlock(&gblcnt_mutex);
29       return 0;
30     }
31   }
32   globalcount += delta;
33   balance_count();
34   spin_unlock(&gblcnt_mutex);
35   return 1;
36 }
```

Quick Quiz 5.54: In Listing 5.18, why does line 41 resend the signal? ■

Lines 51–63 show `balance_count()`, which is similar to that of earlier examples.

Listing 5.19 shows the `add_count()` function. The fastpath spans lines 5–20, and the slowpath lines 21–35. Line 5 sets the per-thread `counting` variable to 1 so that any subsequent signal handlers interrupting this thread will set the `theft` state to ACK rather than READY, allowing this fastpath to complete properly. Line 6 prevents the compiler from reordering any of the fastpath body to precede the setting of `counting`. Lines 7 and 8 check to see if the per-thread data can accommodate the `add_count()` and if there is no ongoing theft in progress, and if so line 9 does the fastpath addition and line 10 notes that the fastpath was taken.

In either case, line 12 prevents the compiler from reordering the fastpath body to follow line 13, which permits any subsequent signal handlers to undertake theft. Line 14 again disables compiler reordering, and then line 15 checks to see if the signal handler deferred the `theft` state-change to READY, and, if so, line 16 executes

Listing 5.20: Signal-Theft Limit Counter Subtract Function

```
1  int sub_count(unsigned long delta)
2  {
3    int fastpath = 0;
4
5    WRITE_ONCE(counting, 1);
6    barrier();
7    if (READ_ONCE(theft) <= THEFT_REQ &&
8        counter >= delta) {
9      WRITE_ONCE(counter, counter - delta);
10     fastpath = 1;
11   }
12   barrier();
13   WRITE_ONCE(counting, 0);
14   barrier();
15   if (READ_ONCE(theft) == THEFT_ACK) {
16     smp_mb();
17     WRITE_ONCE(theft, THEFT_READY);
18   }
19   if (fastpath)
20     return 1;
21   spin_lock(&gblcnt_mutex);
22   globalize_count();
23   if (globalcount < delta) {
24     flush_local_count();
25     if (globalcount < delta) {
26       spin_unlock(&gblcnt_mutex);
27       return 0;
28     }
29   }
30   globalcount -= delta;
31   balance_count();
32   spin_unlock(&gblcnt_mutex);
33   return 1;
34 }
```

Listing 5.21: Signal-Theft Limit Counter Read Function

```
1  unsigned long read_count(void)
2  {
3    int t;
4    unsigned long sum;
5
6    spin_lock(&gblcnt_mutex);
7    sum = globalcount;
8    for_each_thread(t)
9      if (counterp[t] != NULL)
10       sum += READ_ONCE(*counterp[t]);
11   spin_unlock(&gblcnt_mutex);
12   return sum;
13 }
```

a memory barrier to ensure that any CPU that sees line 17 setting state to READY also sees the effects of line 9. If the fastpath addition at line 9 was executed, then line 20 returns success.

Otherwise, we fall through to the slowpath starting at line 21. The structure of the slowpath is similar to those of earlier examples, so its analysis is left as an exercise to the reader. Similarly, the structure of `sub_count()` on Listing 5.20 is the same as that of `add_count()`, so the analysis of `sub_count()` is also left as an exercise for the reader, as is the analysis of `read_count()` in Listing 5.21.

Listing 5.22: Signal-Theft Limit Counter Initialization Functions

```
1  void count_init(void)
2  {
3    struct sigaction sa;
4
5    sa.sa_handler = flush_local_count_sig;
6    sigemptyset(&sa.sa_mask);
7    sa.sa_flags = 0;
8    if (sigaction(SIGUSR1, &sa, NULL) != 0) {
9      perror("sigaction");
10     exit(EXIT_FAILURE);
11   }
12 }
13
14 void count_register_thread(void)
15 {
16   int idx = smp_thread_id();
17
18   spin_lock(&gblcnt_mutex);
19   counterp[idx] = &counter;
20   countermaxp[idx] = &countermax;
21   theftp[idx] = &theft;
22   spin_unlock(&gblcnt_mutex);
23 }
24
25 void count_unregister_thread(int nthreadsexpected)
26 {
27   int idx = smp_thread_id();
28
29   spin_lock(&gblcnt_mutex);
30   globalize_count();
31   counterp[idx] = NULL;
32   countermaxp[idx] = NULL;
33   theftp[idx] = NULL;
34   spin_unlock(&gblcnt_mutex);
35 }
```

Lines 1–12 of Listing 5.22 show `count_init()`, which set up `flush_local_count_sig()` as the signal handler for `SIGUSR1`, enabling the `pthread_kill()` calls in `flush_local_count()` to invoke `flush_local_count_sig()`. The code for thread registry and unregistry is similar to that of earlier examples, so its analysis is left as an exercise for the reader.

5.4.5 Signal-Theft Limit Counter Discussion

The signal-theft implementation runs more than eight times as fast as the atomic implementation on my six-core x86 laptop. Is it always preferable?

The signal-theft implementation would be vastly preferable on Pentium-4 systems, given their slow atomic instructions, but the old 80386-based Sequent Symmetry systems would do much better with the shorter path length of the atomic implementation. However, this increased update-side performance comes at the prices of higher read-side overhead: Those POSIX signals are not free. If ultimate performance is of the essence, you will need to

www.dbooks.org

measure them both on the system that your application is to be deployed on.

Quick Quiz 5.55: Not only are POSIX signals slow, sending one to each thread simply does not scale. What would you do if you had (say) 10,000 threads and needed the read side to be fast? ■

This is but one reason why high-quality APIs are so important: they permit implementations to be changed as required by ever-changing hardware performance characteristics.

Quick Quiz 5.56: What if you want an exact limit counter to be exact only for its lower limit, but to allow the upper limit to be inexact? ■

5.4.6 Applying Exact Limit Counters

Although the exact limit counter implementations presented in this section can be very useful, they are not much help if the counter's value remains near zero at all times, as it might when counting the number of outstanding accesses to an I/O device. The high overhead of such near-zero counting is especially painful given that we normally don't care how many references there are. As noted in the removable I/O device access-count problem posed by Quick Quiz 5.5, the number of accesses is irrelevant except in those rare cases when someone is actually trying to remove the device.

One simple solution to this problem is to add a large "bias" (for example, one billion) to the counter in order to ensure that the value is far enough from zero that the counter can operate efficiently. When someone wants to remove the device, this bias is subtracted from the counter value. Counting the last few accesses will be quite inefficient, but the important point is that the many prior accesses will have been counted at full speed.

Quick Quiz 5.57: What else had you better have done when using a biased counter? ■

Although a biased counter can be quite helpful and useful, it is only a partial solution to the removable I/O device access-count problem called out on page 49. When attempting to remove a device, we must not only know the precise number of current I/O accesses, we also need to prevent any future accesses from starting. One way to accomplish this is to read-acquire a reader-writer lock when updating the counter, and to write-acquire that same reader-writer lock when checking the counter. Code for doing I/O might be as follows:

```
1  read_lock(&mylock);
2  if (removing) {
3    read_unlock(&mylock);
4    cancel_io();
5  } else {
6    add_count(1);
7    read_unlock(&mylock);
8    do_io();
9    sub_count(1);
10 }
```

Line 1 read-acquires the lock, and either line 3 or 7 releases it. Line 2 checks to see if the device is being removed, and, if so, line 3 releases the lock and line 4 cancels the I/O, or takes whatever action is appropriate given that the device is to be removed. Otherwise, line 6 increments the access count, line 7 releases the lock, line 8 performs the I/O, and line 9 decrements the access count.

Quick Quiz 5.58: This is ridiculous! We are *read*-acquiring a reader-writer lock to *update* the counter? What are you playing at??? ■

The code to remove the device might be as follows:

```
1  write_lock(&mylock);
2  removing = 1;
3  sub_count(mybias);
4  write_unlock(&mylock);
5  while (read_count() != 0) {
6    poll(NULL, 0, 1);
7  }
8  remove_device();
```

Line 1 write-acquires the lock and line 4 releases it. Line 2 notes that the device is being removed, and the loop spanning lines 5–7 waits for any I/O operations to complete. Finally, line 8 does any additional processing needed to prepare for device removal.

Quick Quiz 5.59: What other issues would need to be accounted for in a real system? ■

5.5 Parallel Counting Discussion

> This idea that there is generality in the specific is of far-reaching importance.
>
> *Douglas R. Hofstadter*

This chapter has presented the reliability, performance, and scalability problems with traditional counting primitives. The C-language ++ operator is not guaranteed to function reliably in multithreaded code, and atomic operations to a

single variable neither perform nor scale well. This chapter therefore presented a number of counting algorithms that perform and scale extremely well in certain special cases.

It is well worth reviewing the lessons from these counting algorithms. To that end, Section 5.5.1 summarizes performance and scalability, Section 5.5.2 discusses the need for specialization, and finally, Section 5.5.3 enumerates lessons learned and calls attention to later chapters that will expand on these lessons.

5.5.1 Parallel Counting Performance

The top half of Table 5.1 shows the performance of the four parallel statistical counting algorithms. All four algorithms provide near-perfect linear scalability for updates. The per-thread-variable implementation (`count_end.c`) is significantly faster on updates than the array-based implementation (`count_stat.c`), but is slower at reads on large numbers of core, and suffers severe lock contention when there are many parallel readers. This contention can be addressed using the deferred-processing techniques introduced in Chapter 9, as shown on the `count_end_rcu.c` row of Table 5.1. Deferred processing also shines on the `count_stat_eventual.c` row, courtesy of eventual consistency.

Quick Quiz 5.60: On the `count_stat.c` row of Table 5.1, we see that the read-side scales linearly with the number of threads. How is that possible given that the more threads there are, the more per-thread counters must be summed up? ■

Quick Quiz 5.61: Even on the fourth row of Table 5.1, the read-side performance of these statistical counter implementations is pretty horrible. So why bother with them? ■

The bottom half of Table 5.1 shows the performance of the parallel limit-counting algorithms. Exact enforcement of the limits incurs a substantial update-side performance penalty, although on this x86 system that penalty can be reduced by substituting signals for atomic operations. All of these implementations suffer from read-side lock contention in the face of concurrent readers.

Quick Quiz 5.62: Given the performance data shown in the bottom half of Table 5.1, we should always prefer signals over atomic operations, right? ■

Quick Quiz 5.63: Can advanced techniques be applied to address the lock contention for readers seen in the bottom half of Table 5.1? ■

In short, this chapter has demonstrated a number of counting algorithms that perform and scale extremely well in a number of special cases. But must our parallel counting be confined to special cases? Wouldn't it be better to have a general algorithm that operated efficiently in all cases? The next section looks at these questions.

5.5.2 Parallel Counting Specializations

The fact that these algorithms only work well in their respective special cases might be considered a major problem with parallel programming in general. After all, the C-language ++ operator works just fine in single-threaded code, and not just for special cases, but in general, right?

This line of reasoning does contain a grain of truth, but is in essence misguided. The problem is not parallelism as such, but rather scalability. To understand this, first consider the C-language ++ operator. The fact is that it does *not* work in general, only for a restricted range of numbers. If you need to deal with 1,000-digit decimal numbers, the C-language ++ operator will not work for you.

Quick Quiz 5.64: The ++ operator works just fine for 1,000-digit numbers! Haven't you heard of operator overloading??? ■

This problem is not specific to arithmetic. Suppose you need to store and query data. Should you use an ASCII file? XML? A relational database? A linked list? A dense array? A B-tree? A radix tree? Or one of the plethora of other data structures and environments that permit data to be stored and queried? It depends on what you need to do, how fast you need it done, and how large your data set is—even on sequential systems.

Similarly, if you need to count, your solution will depend on how large of numbers you need to work with, how many CPUs need to be manipulating a given number concurrently, how the number is to be used, and what level of performance and scalability you will need.

Nor is this problem specific to software. The design for a bridge meant to allow people to walk across a small brook might be a simple as a single wooden plank. But you would probably not use a plank to span the kilometers-wide mouth of the Columbia River, nor would such a design be advisable for bridges carrying concrete trucks. In short, just as bridge design must change with increasing span and load, so must software design change as the number of CPUs increases. That said, it would be good to automate this process, so that the software adapts to changes in

www.dbooks.org

Table 5.1: Statistical/Limit Counter Performance on x86

Algorithm (`count_*.c`)	Section	Exact?	Updates (ns)	Reads (ns)			
				1 CPU	8 CPUs	64 CPUs	420 CPUs
`stat`	5.2.2		6.3	294	303	315	612
`stat_eventual`	5.2.4		6.4	1	1	1	1
`end`	5.2.3		2.9	301	6,309	147,594	239,683
`end_rcu`	13.5.1		2.9	454	481	508	2,317
`lim`	5.3.2	N	3.2	435	6,678	156,175	239,422
`lim_app`	5.3.4	N	2.4	485	7,041	173,108	239,682
`lim_atomic`	5.4.1	Y	19.7	513	7,085	199,957	239,450
`lim_sig`	5.4.4	Y	4.7	519	6,805	120,000	238,811

hardware configuration and in workload. There has in fact been some research into this sort of automation [AHS+03, SAH+03], and the Linux kernel does some boot-time reconfiguration, including limited binary rewriting. This sort of adaptation will become increasingly important as the number of CPUs on mainstream systems continues to increase.

In short, as discussed in Chapter 3, the laws of physics constrain parallel software just as surely as they constrain mechanical artifacts such as bridges. These constraints force specialization, though in the case of software it might be possible to automate the choice of specialization to fit the hardware and workload in question.

Of course, even generalized counting is quite specialized. We need to do a great number of other things with computers. The next section relates what we have learned from counters to topics taken up later in this book.

5.5.3 Parallel Counting Lessons

The opening paragraph of this chapter promised that our study of counting would provide an excellent introduction to parallel programming. This section makes explicit connections between the lessons from this chapter and the material presented in a number of later chapters.

The examples in this chapter have shown that an important scalability and performance tool is *partitioning*. The counters might be fully partitioned, as in the statistical counters discussed in Section 5.2, or partially partitioned as in the limit counters discussed in Sections 5.3 and 5.4. Partitioning will be considered in far greater depth in Chapter 6, and partial parallelization in particular in Section 6.4, where it is called *parallel fastpath*.

Quick Quiz 5.65: But if we are going to have to partition everything, why bother with shared-memory multithreading? Why not just partition the problem completely and run as multiple processes, each in its own address space? ■

The partially partitioned counting algorithms used locking to guard the global data, and locking is the subject of Chapter 7. In contrast, the partitioned data tended to be fully under the control of the corresponding thread, so that no synchronization whatsoever was required. This *data ownership* will be introduced in Section 6.3.4 and discussed in more detail in Chapter 8.

Because integer addition and subtraction are extremely cheap compared to typical synchronization operations, achieving reasonable scalability requires synchronization operations be used sparingly. One way of achieving this is to batch the addition and subtraction operations, so that a great many of these cheap operations are handled by a single synchronization operation. Batching optimizations of one sort or another are used by each of the counting algorithms listed in Table 5.1.

Finally, the eventually consistent statistical counter discussed in Section 5.2.4 showed how deferring activity (in that case, updating the global counter) can provide substantial performance and scalability benefits. This approach allows common case code to use much cheaper synchronization operations than would otherwise be possible. Chapter 9 will examine a number of additional ways that deferral can improve performance, scalability, and even real-time response.

Summarizing the summary:

1. Partitioning promotes performance and scalability.
2. Partial partitioning, that is, partitioning applied only to common code paths, works almost as well.

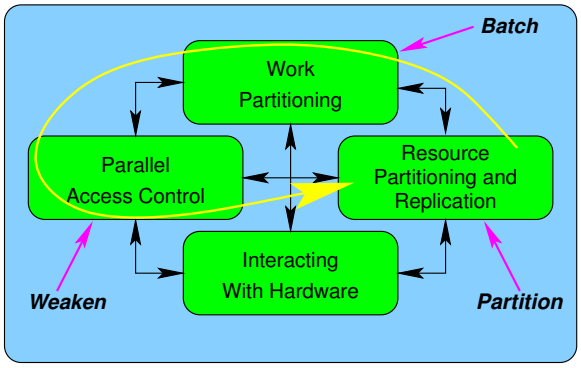


Figure 5.8: Optimization and the Four Parallel-Programming Tasks

3. Partial partitioning can be applied to code (as in Section 5.2's statistical counters' partitioned updates and non-partitioned reads), but also across time (as in Section 5.3's and Section 5.4's limit counters running fast when far from the limit, but slowly when close to the limit).

4. Partitioning across time often batches updates locally in order to reduce the number of expensive global operations, thereby decreasing synchronization overhead, in turn improving performance and scalability. All the algorithms shown in Table 5.1 make heavy use of batching.

5. Read-only code paths should remain read-only: Spurious synchronization writes to shared memory kill performance and scalability, as seen in the `count_end.c` row of Table 5.1.

6. Judicious use of delay promotes performance and scalability, as seen in Section 5.2.4.

7. Parallel performance and scalability is usually a balancing act: Beyond a certain point, optimizing some code paths will degrade others. The `count_stat.c` and `count_end_rcu.c` rows of Table 5.1 illustrate this point.

8. Different levels of performance and scalability will affect algorithm and data-structure design, as do a large number of other factors. Figure 5.1 illustrates this point: Atomic increment might be completely acceptable for a two-CPU system, but be completely inadequate for an eight-CPU system.

Summarizing still further, we have the "big three" methods of increasing performance and scalability, namely (1) *partitioning* over CPUs or threads, (2) *batching* so that more work can be done by each expensive synchronization operations, and (3) *weakening* synchronization operations where feasible. As a rough rule of thumb, you should apply these methods in this order, as was noted earlier in the discussion of Figure 2.6 on page 15. The partitioning optimization applies to the "Resource Partitioning and Replication" bubble, the batching optimization to the "Work Partitioning" bubble, and the weakening optimization to the "Parallel Access Control" bubble, as shown in Figure 5.8. Of course, if you are using special-purpose hardware such as digital signal processors (DSPs), field-programmable gate arrays (FPGAs), or general-purpose graphical processing units (GPGPUs), you may need to pay close attention to the "Interacting With Hardware" bubble throughout the design process. For example, the structure of a GPGPU's hardware threads and memory connectivity might richly reward very careful partitioning and batching design decisions.

In short, as noted at the beginning of this chapter, the simplicity of counting have allowed us to explore many fundamental concurrency issues without the distraction of complex synchronization primitives or elaborate data structures. Such synchronization primitives and data structures are covered in later chapters.

www.dbooks.org

Divide and rule.

Philip II of Macedon

Chapter 6

Partitioning and Synchronization Design

This chapter describes how to design software to take advantage of modern commodity multicore systems by using idioms, or "design patterns" [Ale79, GHJV95, SSRB00], to balance performance, scalability, and response time. Correctly partitioned problems lead to simple, scalable, and high-performance solutions, while poorly partitioned problems result in slow and complex solutions. This chapter will help you design partitioning into your code, with some discussion of batching and weakening as well. The word "design" is very important: You should partition first, batch second, weaken third, and code fourth. Changing this order often leads to poor performance and scalability along with great frustration.[1]

To this end, Section 6.1 presents partitioning exercises, Section 6.2 reviews partitionability design criteria, Section 6.3 discusses synchronization granularity selection, Section 6.4 overviews important parallel-fastpath design patterns that provide speed and scalability on common-case fastpaths while using simpler less-scalable "slow path" fallbacks for unusual situations, and finally Section 6.5 takes a brief look beyond partitioning.

6.1 Partitioning Exercises

> Whenever a theory appears to you as the only possible one, take this as a sign that you have neither understood the theory nor the problem which it was intended to solve.
>
> *Karl Popper*

Although partitioning is more widely understood than it was in the early 2000s, its value is still underappreciated.

[1] That other great dodge around the Laws of Physics, read-only replication, is covered in Chapter 9.

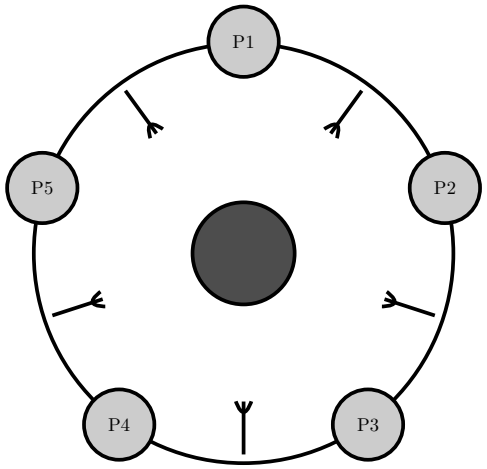


Figure 6.1: Dining Philosophers Problem

Section 6.1.1 therefore takes more highly parallel look at the classic Dining Philosophers problem and Section 6.1.2 revisits the double-ended queue.

6.1.1 Dining Philosophers Problem

Figure 6.1 shows a diagram of the classic Dining Philosophers problem [Dij71]. This problem features five philosophers who do nothing but think and eat a "very difficult kind of spaghetti" which requires two forks to eat.[2] A given philosopher is permitted to use only the forks to his or her immediate right and left, but will not put a given fork down until sated.

The object is to construct an algorithm that, quite literally, prevents starvation. One starvation scenario would be if all of the philosophers picked up their leftmost forks simultaneously. Because none of them will put down

[2] But feel free to instead think in terms of chopsticks.

Edition.2-rc11

www.dbooks.org

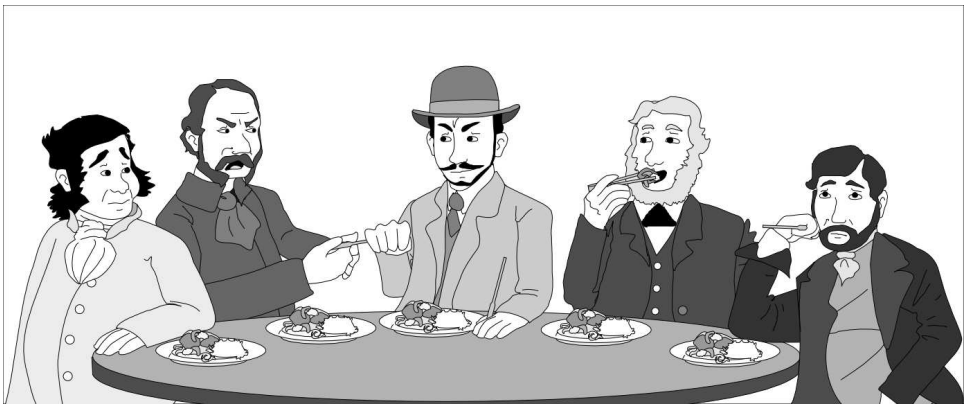


Figure 6.2: Partial Starvation Is Also Bad

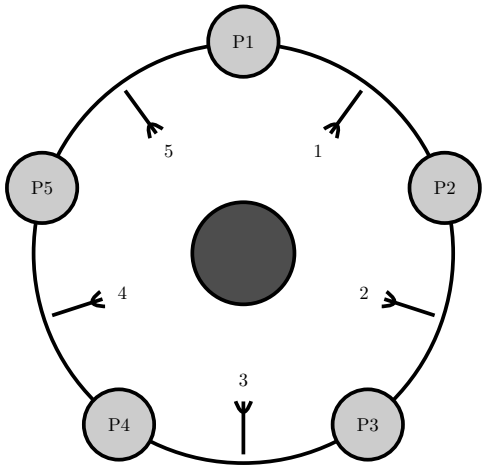


Figure 6.3: Dining Philosophers Problem, Textbook Solution

their fork until after they finished eating, and because none of them may pick up their second fork until at least one of them has finished eating, they all starve. Please note that it is not sufficient to allow at least one philosopher to eat. As Figure 6.2 shows, starvation of even a few of the philosophers is to be avoided.

Dijkstra's solution used a global semaphore, which works fine assuming negligible communications delays, an assumption that became invalid in the late 1980s or early 1990s.[3] More recent solutions number the forks as shown in Figure 6.3. Each philosopher picks up the lowest-numbered fork next to his or her plate, then picks up the other fork. The philosopher sitting in the uppermost position in the diagram thus picks up the leftmost fork first, then the rightmost fork, while the rest of the philosophers instead pick up their rightmost fork first. Because two of the philosophers will attempt to pick up fork 1 first, and because only one of those two philosophers will succeed, there will be five forks available to four philosophers. At least one of these four will have two forks, and will thus be able to eat.

This general technique of numbering resources and acquiring them in numerical order is heavily used as a deadlock-prevention technique. However, it is easy to imagine a sequence of events that will result in only one philosopher eating at a time even though all are hungry:

1. P2 picks up fork 1, preventing P1 from taking a fork.
2. P3 picks up fork 2.
3. P4 picks up fork 3.
4. P5 picks up fork 4.
5. P5 picks up fork 5 and eats.
6. P5 puts down forks 4 and 5.
7. P4 picks up fork 4 and eats.

In short, this algorithm can result in only one philosopher eating at a given time, even when all five philosophers are hungry, despite the fact that there are more than enough forks for two philosophers to eat concurrently. It should be possible to do better than this!

One approach is shown in Figure 6.4, which includes four philosophers rather than five to better illustrate the

[3] It is all too easy to denigrate Dijkstra from the viewpoint of the year 2021, more than 50 years after the fact. If you still feel the need to denigrate Dijkstra, my advice is to publish something, wait 50 years, and then see how well *your* ideas stood the test of time.

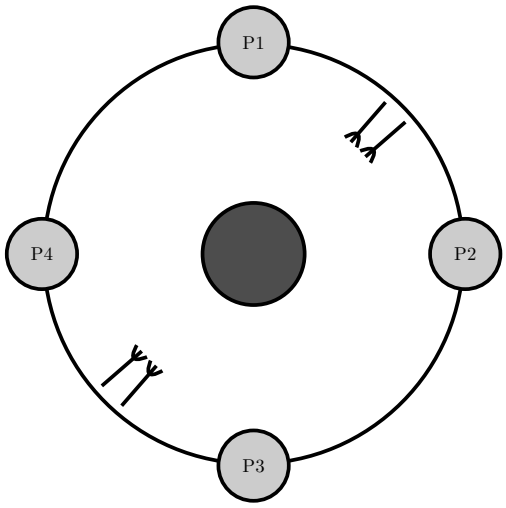


Figure 6.4: Dining Philosophers Problem, Partitioned

partition technique. Here the upper and rightmost philosophers share a pair of forks, while the lower and leftmost philosophers share another pair of forks. If all philosophers are simultaneously hungry, at least two will always be able to eat concurrently. In addition, as shown in the figure, the forks can now be bundled so that the pair are picked up and put down simultaneously, simplifying the acquisition and release algorithms.

Quick Quiz 6.1: Is there a better solution to the Dining Philosophers Problem? ■

This is an example of "horizontal parallelism" [Inm85] or "data parallelism", so named because there is no dependency among the pairs of philosophers. In a horizontally parallel data-processing system, a given item of data would be processed by only one of a replicated set of software components.

Quick Quiz 6.2: And in just what sense can this "horizontal parallelism" be said to be "horizontal"? ■

6.1.2 Double-Ended Queue

A double-ended queue is a data structure containing a list of elements that may be inserted or removed from either end [Knu73]. It has been claimed that a lock-based implementation permitting concurrent operations on both ends of the double-ended queue is difficult [Gro07]. This section shows how a partitioning design strategy can result in a reasonably simple implementation, looking at three general approaches in the following sections.

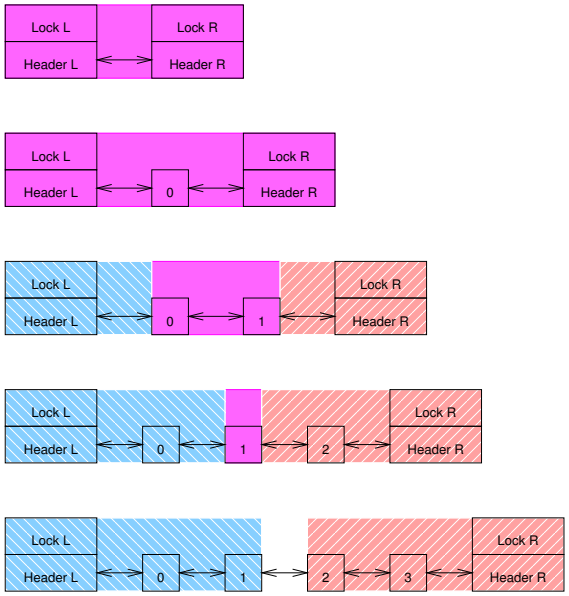


Figure 6.5: Double-Ended Queue With Left- and Right-Hand Locks

6.1.2.1 Left- and Right-Hand Locks

One seemingly straightforward approach would be to use a doubly linked list with a left-hand lock for left-hand-end enqueue and dequeue operations along with a right-hand lock for right-hand-end operations, as shown in Figure 6.5. However, the problem with this approach is that the two locks' domains must overlap when there are fewer than four elements on the list. This overlap is due to the fact that removing any given element affects not only that element, but also its left- and right-hand neighbors. These domains are indicated by color in the figure, with blue with downward stripes indicating the domain of the left-hand lock, red with upward stripes indicating the domain of the right-hand lock, and purple (with no stripes) indicating overlapping domains. Although it is possible to create an algorithm that works this way, the fact that it has no fewer than five special cases should raise a big red flag, especially given that concurrent activity at the other end of the list can shift the queue from one special case to another at any time. It is far better to consider other designs.

6.1.2.2 Compound Double-Ended Queue

One way of forcing non-overlapping lock domains is shown in Figure 6.6. Two separate double-ended queues are run in tandem, each protected by its own lock. This

www.dbooks.org

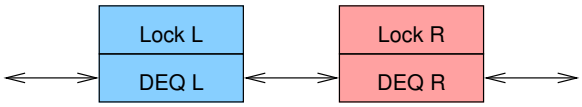


Figure 6.6: Compound Double-Ended Queue

means that elements must occasionally be shuttled from one of the double-ended queues to the other, in which case both locks must be held. A simple lock hierarchy may be used to avoid deadlock, for example, always acquiring the left-hand lock before acquiring the right-hand lock. This will be much simpler than applying two locks to the same double-ended queue, as we can unconditionally left-enqueue elements to the left-hand queue and right-enqueue elements to the right-hand queue. The main complication arises when dequeuing from an empty queue, in which case it is necessary to:

1. If holding the right-hand lock, release it and acquire the left-hand lock.
2. Acquire the right-hand lock.
3. Rebalance the elements across the two queues.
4. Remove the required element if there is one.
5. Release both locks.

Quick Quiz 6.3: In this compound double-ended queue implementation, what should be done if the queue has become non-empty while releasing and reacquiring the lock? ■

The resulting code (`locktdeq.c`) is quite straightforward. The rebalancing operation might well shuttle a given element back and forth between the two queues, wasting time and possibly requiring workload-dependent heuristics to obtain optimal performance. Although this might well be the best approach in some cases, it is interesting to try for an algorithm with greater determinism.

6.1.2.3 Hashed Double-Ended Queue

One of the simplest and most effective ways to deterministically partition a data structure is to hash it. It is possible to trivially hash a double-ended queue by assigning each element a sequence number based on its position in the list, so that the first element left-enqueued into an empty queue is numbered zero and the first element right-enqueued into an empty queue is numbered one. A series of elements left-enqueued into an otherwise-idle queue would be assigned decreasing numbers (-1, -2, -3, ...), while a series of elements right-enqueued into an otherwise-idle queue would be assigned increasing numbers (2, 3, 4, ...). A key point is that it is not necessary to actually represent a given element's number, as this number will be implied by its position in the queue.

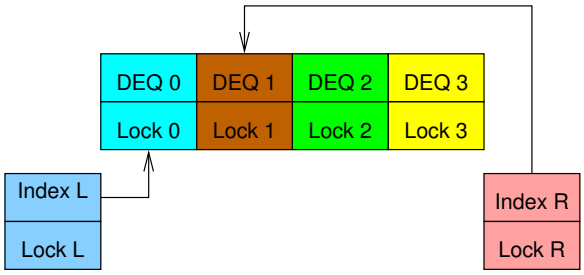


Figure 6.7: Hashed Double-Ended Queue

Given this approach, we assign one lock to guard the left-hand index, one to guard the right-hand index, and one lock for each hash chain. Figure 6.7 shows the resulting data structure given four hash chains. Note that the lock domains do not overlap, and that deadlock is avoided by acquiring the index locks before the chain locks, and by never acquiring more than one lock of a given type (index or chain) at a time.

Each hash chain is itself a double-ended queue, and in this example, each holds every fourth element. The uppermost portion of Figure 6.8 shows the state after a single element ("R_1") has been right-enqueued, with the right-hand index having been incremented to reference hash chain 2. The middle portion of this same figure shows the state after three more elements have been right-enqueued. As you can see, the indexes are back to their initial states (see Figure 6.7), however, each hash chain is now non-empty. The lower portion of this figure shows the state after three additional elements have been left-enqueued and an additional element has been right-enqueued.

From the last state shown in Figure 6.8, a left-dequeue operation would return element "L_{-2}" and leave the left-hand index referencing hash chain 2, which would then contain only a single element ("R_2"). In this state, a left-enqueue running concurrently with a right-enqueue would result in lock contention, but the probability of such contention can be reduced to arbitrarily low levels by using a larger hash table.

Figure 6.9 shows how 16 elements would be organized in a four-hash-bucket parallel double-ended queue. Each underlying single-lock double-ended queue holds a one-quarter slice of the full parallel double-ended queue.

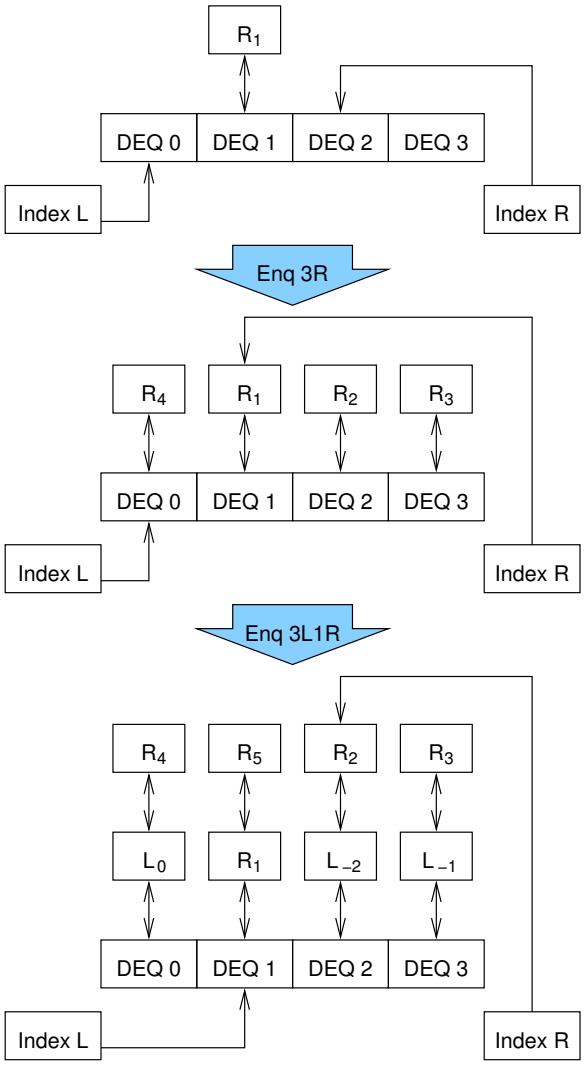


Figure 6.8: Hashed Double-Ended Queue After Insertions

Listing 6.1 shows the corresponding C-language data structure, assuming an existing `struct deq` that provides a trivially locked double-ended-queue implementation. This data structure contains the left-hand lock on line 2, the left-hand index on line 3, the right-hand lock on line 4 (which is cache-aligned in the actual implementation), the right-hand index on line 5, and, finally, the hashed array of simple lock-based double-ended queues on line 6. A high-performance implementation would of course use padding or special alignment directives to avoid false sharing.

Listing 6.1: Lock-Based Parallel Double-Ended Queue Data Structure

```
1 struct pdeq {
2   spinlock_t llock;
3   int lidx;
4   spinlock_t rlock;
5   int ridx;
6   struct deq bkt[PDEQ_N_BKTS];
7 };
```

Listing 6.2: Lock-Based Parallel Double-Ended Queue Implementation

```
1  struct cds_list_head *pdeq_pop_l(struct pdeq *d)
2  {
3    struct cds_list_head *e;
4    int i;
5
6    spin_lock(&d->llock);
7    i = moveright(d->lidx);
8    e = deq_pop_l(&d->bkt[i]);
9    if (e != NULL)
10     d->lidx = i;
11   spin_unlock(&d->llock);
12   return e;
13 }
14
15 struct cds_list_head *pdeq_pop_r(struct pdeq *d)
16 {
17   struct cds_list_head *e;
18   int i;
19
20   spin_lock(&d->rlock);
21   i = moveleft(d->ridx);
22   e = deq_pop_r(&d->bkt[i]);
23   if (e != NULL)
24     d->ridx = i;
25   spin_unlock(&d->rlock);
26   return e;
27 }
28
29 void pdeq_push_l(struct cds_list_head *e, struct pdeq *d)
30 {
31   int i;
32
33   spin_lock(&d->llock);
34   i = d->lidx;
35   deq_push_l(e, &d->bkt[i]);
36   d->lidx = moveleft(d->lidx);
37   spin_unlock(&d->llock);
38 }
39
40 void pdeq_push_r(struct cds_list_head *e, struct pdeq *d)
41 {
42   int i;
43
44   spin_lock(&d->rlock);
45   i = d->ridx;
46   deq_push_r(e, &d->bkt[i]);
47   d->ridx = moveright(d->ridx);
48   spin_unlock(&d->rlock);
49 }
```


www.dbooks.org

R_4	R_5	R_6	R_7
L_0	R_1	R_2	R_3
L_{-4}	L_{-3}	L_{-2}	L_{-1}
L_{-8}	L_{-7}	L_{-6}	L_{-5}

Figure 6.9: Hashed Double-Ended Queue With 16 Elements

Listing 6.2 (`lockhdeq.c`) shows the implementation of the enqueue and dequeue functions.[4] Discussion will focus on the left-hand operations, as the right-hand operations are trivially derived from them.

Lines 1–13 show `pdeq_pop_l()`, which left-dequeues and returns an element if possible, returning `NULL` otherwise. Line 6 acquires the left-hand spinlock, and line 7 computes the index to be dequeued from. Line 8 dequeues the element, and, if line 9 finds the result to be non-`NULL`, line 10 records the new left-hand index. Either way, line 11 releases the lock, and, finally, line 12 returns the element if there was one, or `NULL` otherwise.

Lines 29–38 show `pdeq_push_l()`, which left-enqueues the specified element. Line 33 acquires the left-hand lock, and line 34 picks up the left-hand index. Line 35 left-enqueues the specified element onto the double-ended queue indexed by the left-hand index. Line 36 then updates the left-hand index and line 37 releases the lock.

As noted earlier, the right-hand operations are completely analogous to their left-handed counterparts, so their analysis is left as an exercise for the reader.

Quick Quiz 6.4: Is the hashed double-ended queue a good solution? Why or why not? ■

6.1.2.4 Compound Double-Ended Queue Revisited

This section revisits the compound double-ended queue, using a trivial rebalancing scheme that moves all the elements from the non-empty queue to the now-empty queue.

Quick Quiz 6.5: Move *all* the elements to the queue that became empty? In what possible universe is this brain-dead solution in any way optimal??? ■

[4] One could easily create a polymorphic implementation in any number of languages, but doing so is left as an exercise for the reader.

Listing 6.3: Compound Parallel Double-Ended Queue Implementation

```
1  struct cds_list_head *pdeq_pop_l(struct pdeq *d)
2  {
3    struct cds_list_head *e;
4
5    spin_lock(&d->llock);
6    e = deq_pop_l(&d->ldeq);
7    if (e == NULL) {
8      spin_lock(&d->rlock);
9      e = deq_pop_l(&d->rdeq);
10     cds_list_splice(&d->rdeq.chain, &d->ldeq.chain);
11     CDS_INIT_LIST_HEAD(&d->rdeq.chain);
12     spin_unlock(&d->rlock);
13   }
14   spin_unlock(&d->llock);
15   return e;
16 }
17
18 struct cds_list_head *pdeq_pop_r(struct pdeq *d)
19 {
20   struct cds_list_head *e;
21
22   spin_lock(&d->rlock);
23   e = deq_pop_r(&d->rdeq);
24   if (e == NULL) {
25     spin_unlock(&d->rlock);
26     spin_lock(&d->llock);
27     spin_lock(&d->rlock);
28     e = deq_pop_r(&d->rdeq);
29     if (e == NULL) {
30       e = deq_pop_r(&d->ldeq);
31       cds_list_splice(&d->ldeq.chain, &d->rdeq.chain);
32       CDS_INIT_LIST_HEAD(&d->ldeq.chain);
33     }
34     spin_unlock(&d->llock);
35   }
36   spin_unlock(&d->rlock);
37   return e;
38 }
39
40 void pdeq_push_l(struct cds_list_head *e, struct pdeq *d)
41 {
42   spin_lock(&d->llock);
43   deq_push_l(e, &d->ldeq);
44   spin_unlock(&d->llock);
45 }
46
47 void pdeq_push_r(struct cds_list_head *e, struct pdeq *d)
48 {
49   spin_lock(&d->rlock);
50   deq_push_r(e, &d->rdeq);
51   spin_unlock(&d->rlock);
52 }
```

In contrast to the hashed implementation presented in the previous section, the compound implementation will build on a sequential implementation of a double-ended queue that uses neither locks nor atomic operations.

Listing 6.3 shows the implementation. Unlike the hashed implementation, this compound implementation is asymmetric, so that we must consider the `pdeq_pop_l()` and `pdeq_pop_r()` implementations separately.

Quick Quiz 6.6: Why can't the compound parallel double-ended queue implementation be symmetric? ■

The `pdeq_pop_l()` implementation is shown on lines 1–16 of the figure. Line 5 acquires the left-hand lock, which line 14 releases. Line 6 attempts to left-dequeue an element from the left-hand underlying double-ended queue, and, if successful, skips lines 8–13 to simply return this element. Otherwise, line 8 acquires the right-hand lock, line 9 left-dequeues an element from the right-hand queue, and line 10 moves any remaining elements on the right-hand queue to the left-hand queue, line 11 initializes the right-hand queue, and line 12 releases the right-hand lock. The element, if any, that was dequeued on line 9 will be returned.

The `pdeq_pop_r()` implementation is shown on lines 18–38 of the figure. As before, line 22 acquires the right-hand lock (and line 36 releases it), and line 23 attempts to right-dequeue an element from the right-hand queue, and, if successful, skips lines 25–35 to simply return this element. However, if line 24 determines that there was no element to dequeue, line 25 releases the right-hand lock and lines 26–27 acquire both locks in the proper order. Line 28 then attempts to right-dequeue an element from the right-hand list again, and if line 29 determines that this second attempt has failed, line 30 right-dequeues an element from the left-hand queue (if there is one available), line 31 moves any remaining elements from the left-hand queue to the right-hand queue, and line 32 initializes the left-hand queue. Either way, line 34 releases the left-hand lock.

Quick Quiz 6.7: Why is it necessary to retry the right-dequeue operation on line 28 of Listing 6.3? ■

Quick Quiz 6.8: Surely the left-hand lock must *sometimes* be available!!! So why is it necessary that line 25 of Listing 6.3 unconditionally release the right-hand lock? ■

The `pdeq_push_l()` implementation is shown on lines 40–45 of Listing 6.3. Line 42 acquires the left-hand spinlock, line 43 left-enqueues the element onto the left-hand queue, and finally line 44 releases the lock. The `pdeq_push_r()` implementation (shown on lines 47–52) is quite similar.

Quick Quiz 6.9: But in the case where data is flowing in only one direction, the algorithm shown in Listing 6.3 will have both ends attempting to acquire the same lock whenever the consuming end empties its underlying double-ended queue. Doesn't that mean that sometimes this algorithm fails to provide concurrent access to both ends of the queue even when the queue contains an arbitrarily large number of elements? ■

6.1.2.5 Double-Ended Queue Discussion

The compound implementation is somewhat more complex than the hashed variant presented in Section 6.1.2.3, but is still reasonably simple. Of course, a more intelligent rebalancing scheme could be arbitrarily complex, but the simple scheme shown here has been shown to perform well compared to software alternatives [DCW+11] and even compared to algorithms using hardware assist [DLM+10]. Nevertheless, the best we can hope for from such a scheme is 2x scalability, as at most two threads can be holding the dequeue's locks concurrently. This limitation also applies to algorithms based on non-blocking synchronization, such as the compare-and-swap-based dequeue algorithm of Michael [Mic03].[5]

Quick Quiz 6.10: Why are there not one but two solutions to the double-ended queue problem? ■

In fact, as noted by Dice et al. [DLM+10], an unsynchronized single-threaded double-ended queue significantly outperforms any of the parallel implementations they studied. Therefore, the key point is that there can be significant overhead enqueuing to or dequeuing from a shared queue, regardless of implementation. This should come as no surprise in light of the material in Chapter 3, given the strict first-in-first-out (FIFO) nature of these queues.

Furthermore, these strict FIFO queues are strictly FIFO only with respect to *linearization points* [HW90][6] that are not visible to the caller, in fact, in these examples, the linearization points are buried in the lock-based critical sections. These queues are not strictly FIFO with respect to (say) the times at which the individual operations started [HKLP12]. This indicates that the strict FIFO property is not all that valuable in concurrent programs, and in fact, Kirsch et al. present less-strict queues that provide improved performance and scalability [KLP12].[7] All that said, if you are pushing all the data used by your concurrent program through a single queue, you really need to rethink your overall design.

[5] This paper is interesting in that it showed that special double-compare-and-swap (DCAS) instructions are not needed for lock-free implementations of double-ended queues. Instead, the common compare-and-swap (e.g., x86 cmpxchg) suffices.

[6] In short, a linearization point is a single point within a given function where that function can be said to have taken effect. In this lock-based implementation, the linearization points can be said to be anywhere within the critical section that does the work.

[7] Nir Shavit produced relaxed stacks for roughly the same reasons [Sha11]. This situation leads some to believe that the linearization points are useful to theorists rather than developers, and leads others to wonder to what extent the designers of such data structures and algorithms were considering the needs of their users.

www.dbooks.org

6.1.3 Partitioning Example Discussion

The optimal solution to the dining philosophers problem given in the answer to the Quick Quiz in Section 6.1.1 is an excellent example of "horizontal parallelism" or "data parallelism". The synchronization overhead in this case is nearly (or even exactly) zero. In contrast, the double-ended queue implementations are examples of "vertical parallelism" or "pipelining", given that data moves from one thread to another. The tighter coordination required for pipelining in turn requires larger units of work to obtain a given level of efficiency.

Quick Quiz 6.11: The tandem double-ended queue runs about twice as fast as the hashed double-ended queue, even when I increase the size of the hash table to an insanely large number. Why is that? ■

Quick Quiz 6.12: Is there a significantly better way of handling concurrency for double-ended queues? ■

These two examples show just how powerful partitioning can be in devising parallel algorithms. Section 6.3.5 looks briefly at a third example, matrix multiply. However, all three of these examples beg for more and better design criteria for parallel programs, a topic taken up in the next section.

6.2 Design Criteria

> One pound of learning requires ten pounds of commonsense to apply it.
>
> *Persian proverb*

One way to obtain the best performance and scalability is to simply hack away until you converge on the best possible parallel program. Unfortunately, if your program is other than microscopically tiny, the space of possible parallel programs is so huge that convergence is not guaranteed in the lifetime of the universe. Besides, what exactly is the "best possible parallel program"? After all, Section 2.2 called out no fewer than three parallel-programming goals of performance, productivity, and generality, and the best possible performance will likely come at a cost in terms of productivity and generality. We clearly need to be able to make higher-level choices at design time in order to arrive at an acceptably good parallel program before that program becomes obsolete.

However, more detailed design criteria are required to actually produce a real-world design, a task taken up in this section. This being the real world, these criteria often conflict to a greater or lesser degree, requiring that the designer carefully balance the resulting tradeoffs.

As such, these criteria may be thought of as the "forces" acting on the design, with particularly good tradeoffs between these forces being called "design patterns" [Ale79, GHJV95].

The design criteria for attaining the three parallel-programming goals are speedup, contention, overhead, read-to-write ratio, and complexity:

Speedup: As noted in Section 2.2, increased performance is the major reason to go to all of the time and trouble required to parallelize it. Speedup is defined to be the ratio of the time required to run a sequential version of the program to the time required to run a parallel version.

Contention: If more CPUs are applied to a parallel program than can be kept busy by that program, the excess CPUs are prevented from doing useful work by contention. This may be lock contention, memory contention, or a host of other performance killers.

Work-to-Synchronization Ratio: A uniprocessor, single-threaded, non-preemptible, and non-interruptible[8] version of a given parallel program would not need any synchronization primitives. Therefore, any time consumed by these primitives (including communication cache misses as well as message latency, locking primitives, atomic instructions, and memory barriers) is overhead that does not contribute directly to the useful work that the program is intended to accomplish. Note that the important measure is the relationship between the synchronization overhead and the overhead of the code in the critical section, with larger critical sections able to tolerate greater synchronization overhead. The work-to-synchronization ratio is related to the notion of synchronization efficiency.

Read-to-Write Ratio: A data structure that is rarely updated may often be replicated rather than partitioned, and furthermore may be protected with asymmetric synchronization primitives that reduce readers' synchronization overhead at the expense of that of writers, thereby reducing overall synchronization overhead. Corresponding optimizations are possible for frequently updated data structures, as discussed in Chapter 5.

[8] Either by masking interrupts or by being oblivious to them.

Complexity: A parallel program is more complex than an equivalent sequential program because the parallel program has a much larger state space than does the sequential program, although large state spaces having regular structures can in some cases be easily understood. A parallel programmer must consider synchronization primitives, messaging, locking design, critical-section identification, and deadlock in the context of this larger state space.

This greater complexity often translates to higher development and maintenance costs. Therefore, budgetary constraints can limit the number and types of modifications made to an existing program, since a given degree of speedup is worth only so much time and trouble. Worse yet, added complexity can actually *reduce* performance and scalability.

Therefore, beyond a certain point, there may be potential sequential optimizations that are cheaper and more effective than parallelization. As noted in Section 2.2.1, parallelization is but one performance optimization of many, and is furthermore an optimization that applies most readily to CPU-based bottlenecks.

These criteria will act together to enforce a maximum speedup. The first three criteria are deeply interrelated, so the remainder of this section analyzes these interrelationships.[9]

Note that these criteria may also appear as part of the requirements specification. For example, speedup may act as a relative desideratum ("the faster, the better") or as an absolute requirement of the workload ("the system must support at least 1,000,000 web hits per second"). Classic design pattern languages describe relative desiderata as forces and absolute requirements as context.

An understanding of the relationships between these design criteria can be very helpful when identifying appropriate design tradeoffs for a parallel program.

1. The less time a program spends in exclusive-lock critical sections, the greater the potential speedup. This is a consequence of Amdahl's Law [Amd67] because only one CPU may execute within a given exclusive-lock critical section at a given time.

   More specifically, for unbounded linear scalability, the fraction of time that the program spends in a given exclusive critical section must decrease as the number of CPUs increases. For example, a program will not scale to 10 CPUs unless it spends much less than one tenth of its time in the most-restrictive exclusive-lock critical section.

2. Contention effects consume the excess CPU and/or wallclock time when the actual speedup is less than the number of available CPUs. The larger the gap between the number of CPUs and the actual speedup, the less efficiently the CPUs will be used. Similarly, the greater the desired efficiency, the smaller the achievable speedup.

3. If the available synchronization primitives have high overhead compared to the critical sections that they guard, the best way to improve speedup is to reduce the number of times that the primitives are invoked. This can be accomplished by batching critical sections, using data ownership (see Chapter 8), using asymmetric primitives (see Section 9), or by using a coarse-grained design such as code locking.

4. If the critical sections have high overhead compared to the primitives guarding them, the best way to improve speedup is to increase parallelism by moving to reader/writer locking, data locking, asymmetric, or data ownership.

5. If the critical sections have high overhead compared to the primitives guarding them and the data structure being guarded is read much more often than modified, the best way to increase parallelism is to move to reader/writer locking or asymmetric primitives.

6. Many changes that improve SMP performance, for example, reducing lock contention, also improve real-time latencies [McK05c].

Quick Quiz 6.13: Don't all these problems with critical sections mean that we should just always use non-blocking synchronization [Her90], which don't have critical sections? ■

It is worth reiterating that contention has many guises, including lock contention, memory contention, cache overflow, thermal throttling, and much else besides. This chapter looks primarily at lock and memory contention.

[9] A real-world parallel system will be subject to many additional design criteria, such as data-structure layout, memory size, memory-hierarchy latencies, bandwidth limitations, and I/O issues.

www.dbooks.org

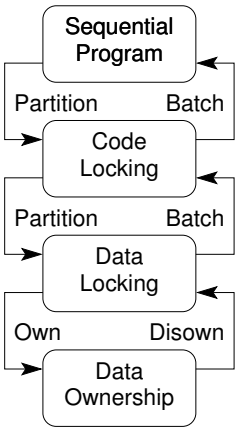


Figure 6.10: Design Patterns and Lock Granularity

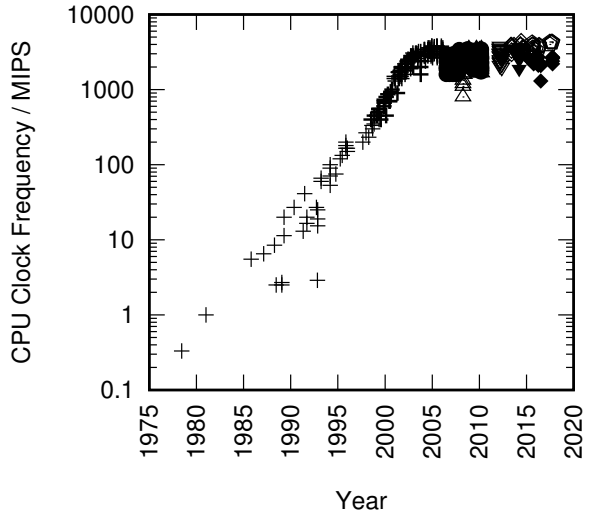


Figure 6.11: MIPS/Clock-Frequency Trend for Intel CPUs

6.3 Synchronization Granularity

> Doing little things well is a step toward doing big things better.
>
> *Harry F. Banks*

Figure 6.10 gives a pictorial view of different levels of synchronization granularity, each of which is described in one of the following sections. These sections focus primarily on locking, but similar granularity issues arise with all forms of synchronization.

6.3.1 Sequential Program

If the program runs fast enough on a single processor, and has no interactions with other processes, threads, or interrupt handlers, you should remove the synchronization primitives and spare yourself their overhead and complexity. Some years back, there were those who would argue that Moore's Law would eventually force all programs into this category. However, as can be seen in Figure 6.11, the exponential increase in single-threaded performance halted in about 2003. Therefore, increasing performance will increasingly require parallelism.[10] Given that back in 2006 Paul typed the first version of this sentence on a dual-core laptop, and further given that many of the graphs added in 2020 were generated on a system with 56 hardware threads per socket, parallelism is well and truly here. It is also important to note that Ethernet bandwidth is continuing to grow, as shown in Figure 6.12. This growth will continue to motivate multithreaded servers in order to handle the communications load.

Please note that this does *not* mean that you should code each and every program in a multi-threaded manner. Again, if a program runs quickly enough on a single processor, spare yourself the overhead and complexity of SMP synchronization primitives. The simplicity of the hash-table lookup code in Listing 6.4 underscores this point.[11] A key point is that speedups due to parallelism are normally limited to the number of CPUs. In contrast, speedups due to sequential optimizations, for example, careful choice of data structure, can be arbitrarily large.

On the other hand, if you are not in this happy situation, read on!

6.3.2 Code Locking

Code locking is quite simple due to the fact that is uses only global locks.[12] It is especially easy to retrofit an existing program to use code locking in order to run it on a multiprocessor. If the program has only a single shared resource, code locking will even give optimal performance.

[10] This plot shows clock frequencies for newer CPUs theoretically capable of retiring one or more instructions per clock, and MIPS for older CPUs requiring multiple clocks to execute even the simplest instruction. The reason for taking this approach is that the newer CPUs' ability to retire multiple instructions per clock is typically limited by memory-system performance.

[11] The examples in this section are taken from Hart et al. [HMB06], adapted for clarity by gathering related code from multiple files.

[12] If your program instead has locks in data structures, or, in the case of Java, uses classes with synchronized instances, you are instead using "data locking", described in Section 6.3.3.

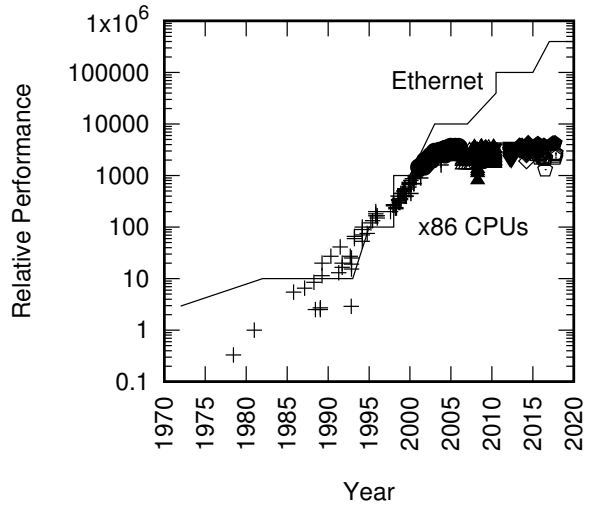


Figure 6.12: Ethernet Bandwidth vs. Intel x86 CPU Performance

Listing 6.4: Sequential-Program Hash Table Search

```
1  struct hash_table
2  {
3    long nbuckets;
4    struct node **buckets;
5  };
6
7  typedef struct node {
8    unsigned long key;
9    struct node *next;
10 } node_t;
11
12 int hash_search(struct hash_table *h, long key)
13 {
14   struct node *cur;
15
16   cur = h->bucketskey % h->nbuckets;
17   while (cur != NULL) {
18     if (cur->key >= key) {
19       return (cur->key == key);
20     }
21     cur = cur->next;
22   }
23   return 0;
24 }
```

However, many of the larger and more complex programs require much of the execution to occur in critical sections, which in turn causes code locking to sharply limits their scalability.

Therefore, you should use code locking on programs that spend only a small fraction of their execution time in critical sections or from which only modest scaling is required. In these cases, code locking will provide a relatively simple program that is very similar to its sequential counterpart, as can be seen in Listing 6.5. However, note that the simple return of the comparison in `hash_search()` in Listing 6.4 has now become three statements due to the need to release the lock before returning.

Listing 6.5: Code-Locking Hash Table Search

```
1  spinlock_t hash_lock;
2
3  struct hash_table
4  {
5    long nbuckets;
6    struct node **buckets;
7  };
8
9  typedef struct node {
10   unsigned long key;
11   struct node *next;
12 } node_t;
13
14 int hash_search(struct hash_table *h, long key)
15 {
16   struct node *cur;
17   int retval;
18
19   spin_lock(&hash_lock);
20   cur = h->bucketskey % h->nbuckets;
21   while (cur != NULL) {
22     if (cur->key >= key) {
23       retval = (cur->key == key);
24       spin_unlock(&hash_lock);
25       return retval;
26     }
27     cur = cur->next;
28   }
29   spin_unlock(&hash_lock);
30   return 0;
31 }
```

Unfortunately, code locking is particularly prone to "lock contention", where multiple CPUs need to acquire the lock concurrently. SMP programmers who have taken care of groups of small children (or groups of older people who are acting like children) will immediately recognize the danger of having only one of something, as illustrated in Figure 6.13.

One solution to this problem, named "data locking", is described in the next section.

www.dbooks.org

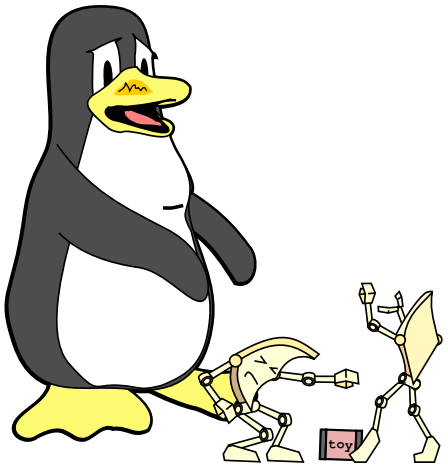


Figure 6.13: Lock Contention

6.3.3 Data Locking

Many data structures may be partitioned, with each partition of the data structure having its own lock. Then the critical sections for each part of the data structure can execute in parallel, although only one instance of the critical section for a given part could be executing at a given time. You should use data locking when contention must be reduced, and where synchronization overhead is not limiting speedups. Data locking reduces contention by distributing the instances of the overly-large critical section across multiple data structures, for example, maintaining per-hash-bucket critical sections in a hash table, as shown in Listing 6.6. The increased scalability again results in a slight increase in complexity in the form of an additional data structure, the `struct bucket`.

In contrast with the contentious situation shown in Figure 6.13, data locking helps promote harmony, as illustrated by Figure 6.14—and in parallel programs, this *almost* always translates into increased performance and scalability. For this reason, data locking was heavily used by Sequent in its kernels [BK85, Inm85, Gar90, Dov90, MD92, MG92, MS93].

However, as those who have taken care of small children can again attest, even providing enough to go around is no guarantee of tranquillity. The analogous situation can arise in SMP programs. For example, the Linux kernel maintains a cache of files and directories (called "dcache"). Each entry in this cache has its own lock, but the entries corresponding to the root directory and its direct descendants are much more likely to be traversed than

Listing 6.6: Data-Locking Hash Table Search

```
1  struct hash_table
2  {
3    long nbuckets;
4    struct bucket **buckets;
5  };
6
7  struct bucket {
8    spinlock_t bucket_lock;
9    node_t *list_head;
10 };
11
12 typedef struct node {
13   unsigned long key;
14   struct node *next;
15 } node_t;
16
17 int hash_search(struct hash_table *h, long key)
18 {
19   struct bucket *bp;
20   struct node *cur;
21   int retval;
22
23   bp = h->buckets[key % h->nbuckets];
24   spin_lock(&bp->bucket_lock);
25   cur = bp->list_head;
26   while (cur != NULL) {
27     if (cur->key >= key) {
28       retval = (cur->key == key);
29       spin_unlock(&bp->bucket_lock);
30       return retval;
31     }
32     cur = cur->next;
33   }
34   spin_unlock(&bp->bucket_lock);
35   return 0;
36 }
```

are more obscure entries. This can result in many CPUs contending for the locks of these popular entries, resulting in a situation not unlike that shown in Figure 6.15.

In many cases, algorithms can be designed to reduce the instance of data skew, and in some cases eliminate it entirely (for example, in the Linux kernel's dcache [MSS04, Cor10a, Bro15a, Bro15b, Bro15c]). Data locking is often used for partitionable data structures such as hash tables, as well as in situations where multiple entities are each represented by an instance of a given data structure. The Linux-kernel task list is an example of the latter, each task structure having its own `alloc_lock` and `pi_lock`.

A key challenge with data locking on dynamically allocated structures is ensuring that the structure remains in existence while the lock is being acquired [GKAS99]. The code in Listing 6.6 finesses this challenge by placing the locks in the statically allocated hash buckets, which are never freed. However, this trick would not work if the hash table were resizeable, so that the locks were now dynamically allocated. In this case, there would need to

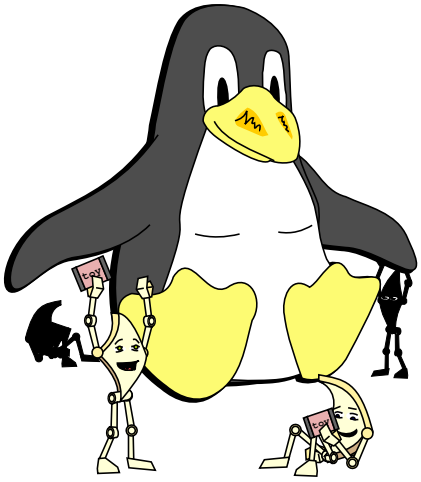


Figure 6.14: Data Locking

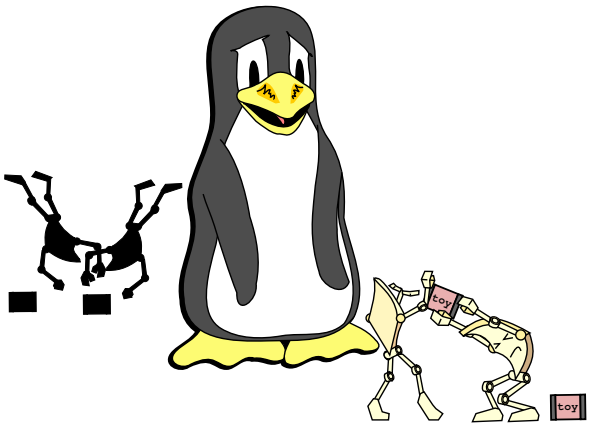


Figure 6.15: Data Locking and Skew

be some means to prevent the hash bucket from being freed during the time that its lock was being acquired.

Quick Quiz 6.14: What are some ways of preventing a structure from being freed while its lock is being acquired? ■

6.3.4 Data Ownership

Data ownership partitions a given data structure over the threads or CPUs, so that each thread/CPU accesses its subset of the data structure without any synchronization overhead whatsoever. However, if one thread wishes to access some other thread's data, the first thread is unable to do so directly. Instead, the first thread must communicate with the second thread, so that the second thread performs the operation on behalf of the first, or, alternatively, migrates the data to the first thread.

Data ownership might seem arcane, but it is used very frequently:

1. Any variables accessible by only one CPU or thread (such as `auto` variables in C and C++) are owned by that CPU or process.

2. An instance of a user interface owns the corresponding user's context. It is very common for applications interacting with parallel database engines to be written as if they were entirely sequential programs. Such applications own the user interface and his current action. Explicit parallelism is thus confined to the database engine itself.

3. Parametric simulations are often trivially parallelized by granting each thread ownership of a particular region of the parameter space. There are also computing frameworks designed for this type of problem [Uni08a].

If there is significant sharing, communication between the threads or CPUs can result in significant complexity and overhead. Furthermore, if the most-heavily used data happens to be that owned by a single CPU, that CPU will be a "hot spot", sometimes with results resembling that shown in Figure 6.15. However, in situations where no sharing is required, data ownership achieves ideal performance, and with code that can be as simple as the sequential-program case shown in Listing 6.4. Such situations are often referred to as "embarrassingly parallel", and, in the best case, resemble the situation previously shown in Figure 6.14.

Another important instance of data ownership occurs when the data is read-only, in which case, all threads can "own" it via replication.

Data ownership will be presented in more detail in Chapter 8.

6.3.5 Locking Granularity and Performance

This section looks at locking granularity and performance from a mathematical synchronization-efficiency viewpoint. Readers who are uninspired by mathematics might choose to skip this section.

www.dbooks.org

The approach is to use a crude queueing model for the efficiency of synchronization mechanism that operate on a single shared global variable, based on an M/M/1 queue. M/M/1 queuing models are based on an exponentially distributed "inter-arrival rate" λ and an exponentially distributed "service rate" μ. The inter-arrival rate λ can be thought of as the average number of synchronization operations per second that the system would process if the synchronization were free, in other words, λ is an inverse measure of the overhead of each non-synchronization unit of work. For example, if each unit of work was a transaction, and if each transaction took one millisecond to process, excluding synchronization overhead, then λ would be 1,000 transactions per second.

The service rate μ is defined similarly, but for the average number of synchronization operations per second that the system would process if the overhead of each transaction was zero, and ignoring the fact that CPUs must wait on each other to complete their synchronization operations, in other words, μ can be roughly thought of as the synchronization overhead in absence of contention. For example, suppose that each transaction's synchronization operation involves an atomic increment instruction, and that a computer system is able to do a private-variable atomic increment every 5 nanoseconds on each CPU (see Figure 5.1).[13] The value of μ is therefore about 200,000,000 atomic increments per second.

Of course, the value of λ increases as increasing numbers of CPUs increment a shared variable because each CPU is capable of processing transactions independently (again, ignoring synchronization):

$$\lambda = n\lambda_0 \tag{6.1}$$

Here, n is the number of CPUs and λ_0 is the transaction-processing capability of a single CPU. Note that the expected time for a single CPU to execute a single transaction in the absence of contention is $1/\lambda_0$.

Because the CPUs have to "wait in line" behind each other to get their chance to increment the single shared variable, we can use the M/M/1 queueing-model expression for the expected total waiting time:

$$T = \frac{1}{\mu - \lambda} \tag{6.2}$$

[13] Of course, if there are 8 CPUs all incrementing the same shared variable, then each CPU must wait at least 35 nanoseconds for each of the other CPUs to do its increment before consuming an additional 5 nanoseconds doing its own increment. In fact, the wait will be longer due to the need to move the variable from one CPU to another.

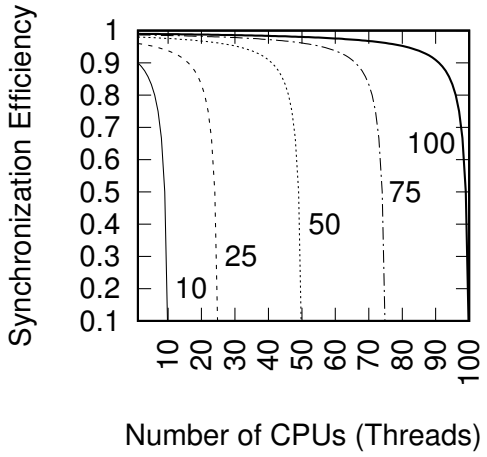


Figure 6.16: Synchronization Efficiency

Substituting the above value of λ:

$$T = \frac{1}{\mu - n\lambda_0} \tag{6.3}$$

Now, the efficiency is just the ratio of the time required to process a transaction in absence of synchronization ($1/\lambda_0$) to the time required including synchronization ($T + 1/\lambda_0$):

$$e = \frac{1/\lambda_0}{T + 1/\lambda_0} \tag{6.4}$$

Substituting the above value for T and simplifying:

$$e = \frac{\frac{\mu}{\lambda_0} - n}{\frac{\mu}{\lambda_0} - (n-1)} \tag{6.5}$$

But the value of μ/λ_0 is just the ratio of the time required to process the transaction (absent synchronization overhead) to that of the synchronization overhead itself (absent contention). If we call this ratio f, we have:

$$e = \frac{f - n}{f - (n-1)} \tag{6.6}$$

Figure 6.16 plots the synchronization efficiency e as a function of the number of CPUs/threads n for a few values of the overhead ratio f. For example, again using the 5-nanosecond atomic increment, the $f = 10$ line corresponds to each CPU attempting an atomic increment every 50 nanoseconds, and the $f = 100$ line corresponds to each CPU attempting an atomic increment every 500 nanoseconds, which in turn corresponds to some hundreds (perhaps thousands) of instructions. Given that

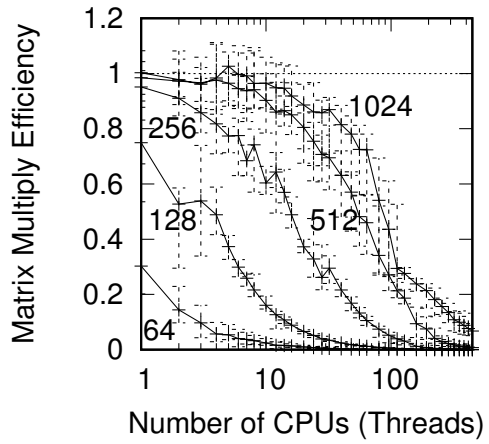


Figure 6.17: Matrix Multiply Efficiency

each trace drops off sharply with increasing numbers of CPUs or threads, we can conclude that synchronization mechanisms based on atomic manipulation of a single global shared variable will not scale well if used heavily on current commodity hardware. This is an abstract mathematical depiction of the forces leading to the parallel counting algorithms that were discussed in Chapter 5. Your real-world mileage may differ.

Nevertheless, the concept of efficiency is useful, and even in cases having little or no formal synchronization. Consider for example a matrix multiply, in which the columns of one matrix are multiplied (via "dot product") by the rows of another, resulting in an entry in a third matrix. Because none of these operations conflict, it is possible to partition the columns of the first matrix among a group of threads, with each thread computing the corresponding columns of the result matrix. The threads can therefore operate entirely independently, with no synchronization overhead whatsoever, as is done in `matmul.c`. One might therefore expect a perfect efficiency of 1.0.

However, Figure 6.17 tells a different story, especially for a 64-by-64 matrix multiply, which never gets above an efficiency of about 0.3, even when running single-threaded, and drops sharply as more threads are added.[14] The 128-by-128 matrix does better, but still fails to demonstrate much performance increase with added threads. The 256-by-256 matrix does scale reasonably well, but only up to a handful of CPUs. The 512-by-512 matrix multiply's efficiency is measurably less than 1.0 on as few as 10 threads, and even the 1024-by-1024 matrix multiply deviates noticeably from perfection at a few tens of threads. Nevertheless, this figure clearly demonstrates the performance and scalability benefits of batching: If you must incur synchronization overhead, you may as well get your money's worth.

Quick Quiz 6.15: How can a single-threaded 64-by-64 matrix multiple possibly have an efficiency of less than 1.0? Shouldn't all of the traces in Figure 6.17 have efficiency of exactly 1.0 when running on one thread? ■

Given these inefficiencies, it is worthwhile to look into more-scalable approaches such as the data locking described in Section 6.3.3 or the parallel-fastpath approach discussed in the next section.

Quick Quiz 6.16: How are data-parallel techniques going to help with matrix multiply? It is *already* data parallel!!! ■

6.4 Parallel Fastpath

> There are two ways of meeting difficulties: You alter the difficulties, or you alter yourself to meet them.
>
> *Phyllis Bottome*

Fine-grained (and therefore *usually* higher-performance) designs are typically more complex than are coarser-grained designs. In many cases, most of the overhead is incurred by a small fraction of the code [Knu73]. So why not focus effort on that small fraction?

This is the idea behind the parallel-fastpath design pattern, to aggressively parallelize the common-case code path without incurring the complexity that would be required to aggressively parallelize the entire algorithm. You must understand not only the specific algorithm you wish to parallelize, but also the workload that the algorithm will be subjected to. Great creativity and design effort is often required to construct a parallel fastpath.

Parallel fastpath combines different patterns (one for the fastpath, one elsewhere) and is therefore a template pattern. The following instances of parallel fastpath occur often enough to warrant their own patterns, as depicted in Figure 6.18:

1. Reader/Writer Locking (described below in Section 6.4.1).
2. Read-copy update (RCU), which may be used as a high-performance replacement for reader/writer lock-

[14] In contrast to the smooth traces of Figure 6.16, the wide error bars and jagged traces of Figure 6.17 gives evidence of its real-world nature.

www.dbooks.org

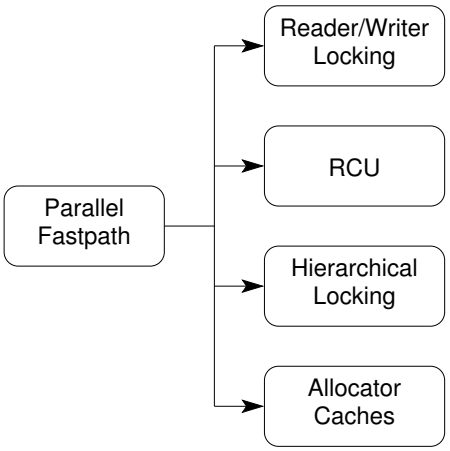


Figure 6.18: Parallel-Fastpath Design Patterns

ing, is introduced in Section 9.5. Other alternatives include hazard pointers (Section 9.3) and sequence locking (Section 9.4). These alternatives will not be discussed further in this chapter.

3. Hierarchical Locking ([McK96a]), which is touched upon in Section 6.4.2.

4. Resource Allocator Caches ([McK96a, MS93]). See Section 6.4.3 for more detail.

6.4.1 Reader/Writer Locking

If synchronization overhead is negligible (for example, if the program uses coarse-grained parallelism with large critical sections), and if only a small fraction of the critical sections modify data, then allowing multiple readers to proceed in parallel can greatly increase scalability. Writers exclude both readers and each other. There are many implementations of reader-writer locking, including the POSIX implementation described in Section 4.2.4. Listing 6.7 shows how the hash search might be implemented using reader-writer locking.

Reader/writer locking is a simple instance of asymmetric locking. Snaman [ST87] describes a more ornate six-mode asymmetric locking design used in several clustered systems. Locking in general and reader-writer locking in particular is described extensively in Chapter 7.

6.4.2 Hierarchical Locking

The idea behind hierarchical locking is to have a coarse-grained lock that is held only long enough to work out which fine-grained lock to acquire. Listing 6.8 shows how our hash-table search might be adapted to do hierarchical locking, but also shows the great weakness of this approach: we have paid the overhead of acquiring a second lock, but we only hold it for a short time. In this case, the simpler data-locking approach would be simpler and likely perform better.

Listing 6.7: Reader-Writer-Locking Hash Table Search

```
1  rwlock_t hash_lock;
2
3  struct hash_table
4  {
5    long nbuckets;
6    struct node **buckets;
7  };
8
9  typedef struct node {
10   unsigned long key;
11   struct node *next;
12 } node_t;
13
14 int hash_search(struct hash_table *h, long key)
15 {
16   struct node *cur;
17   int retval;
18
19   read_lock(&hash_lock);
20   cur = h->bucketskey % h->nbuckets;
21   while (cur != NULL) {
22     if (cur->key >= key) {
23       retval = (cur->key == key);
24       read_unlock(&hash_lock);
25       return retval;
26     }
27     cur = cur->next;
28   }
29   read_unlock(&hash_lock);
30   return 0;
31 }
```

Quick Quiz 6.17: In what situation would hierarchical locking work well? ■

6.4.3 Resource Allocator Caches

This section presents a simplified schematic of a parallel fixed-block-size memory allocator. More detailed descriptions may be found in the literature [MG92, MS93, BA01, MSK01, Eva11, Ken20] or in the Linux kernel [Tor03].

6.4.3.1 Parallel Resource Allocation Problem

The basic problem facing a parallel memory allocator is the tension between the need to provide extremely fast memory allocation and freeing in the common case and the need to efficiently distribute memory in face of unfavorable allocation and freeing patterns.

Listing 6.8: Hierarchical-Locking Hash Table Search

```
1  struct hash_table
2  {
3    long nbuckets;
4    struct bucket **buckets;
5  };
6
7  struct bucket {
8    spinlock_t bucket_lock;
9    node_t *list_head;
10 };
11
12 typedef struct node {
13   spinlock_t node_lock;
14   unsigned long key;
15   struct node *next;
16 } node_t;
17
18 int hash_search(struct hash_table *h, long key)
19 {
20   struct bucket *bp;
21   struct node *cur;
22   int retval;
23
24   bp = h->bucketskey % h->nbuckets;
25   spin_lock(&bp->bucket_lock);
26   cur = bp->list_head;
27   while (cur != NULL) {
28     if (cur->key >= key) {
29       spin_lock(&cur->node_lock);
30       spin_unlock(&bp->bucket_lock);
31       retval = (cur->key == key);
32       spin_unlock(&cur->node_lock);
33       return retval;
34     }
35     cur = cur->next;
36   }
37   spin_unlock(&bp->bucket_lock);
38   return 0;
39 }
```

To see this tension, consider a straightforward application of data ownership to this problem—simply carve up memory so that each CPU owns its share. For example, suppose that a system with 12 CPUs has 64 gigabytes of memory, for example, the laptop I am using right now. We could simply assign each CPU a five-gigabyte region of memory, and allow each CPU to allocate from its own region, without the need for locking and its complexities and overheads. Unfortunately, this scheme fails when CPU 0 only allocates memory and CPU 1 only frees it, as happens in simple producer-consumer workloads.

The other extreme, code locking, suffers from excessive lock contention and overhead [MS93].

6.4.3.2 Parallel Fastpath for Resource Allocation

The commonly used solution uses parallel fastpath with each CPU owning a modest cache of blocks, and with a large code-locked shared pool for additional blocks. To prevent any given CPU from monopolizing the memory blocks, we place a limit on the number of blocks that can

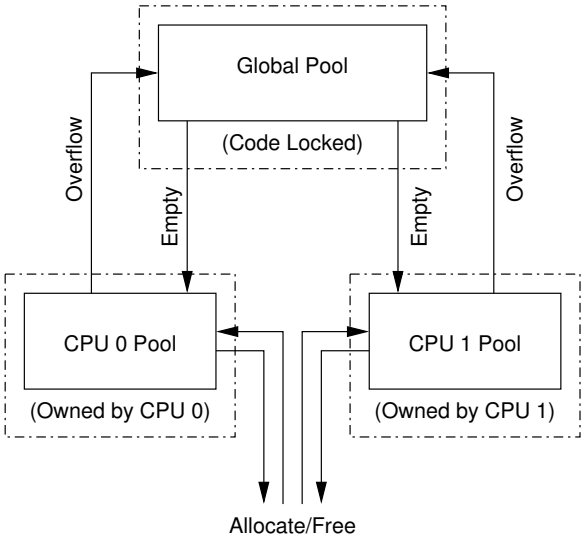


Figure 6.19: Allocator Cache Schematic

be in each CPU's cache. In a two-CPU system, the flow of memory blocks will be as shown in Figure 6.19: when a given CPU is trying to free a block when its pool is full, it sends blocks to the global pool, and, similarly, when that CPU is trying to allocate a block when its pool is empty, it retrieves blocks from the global pool.

6.4.3.3 Data Structures

The actual data structures for a "toy" implementation of allocator caches are shown in Listing 6.9. The "Global Pool" of Figure 6.19 is implemented by `globalmem` of type `struct globalmempool`, and the two CPU pools by the per-thread variable `perthreadmem` of type `struct perthreadmempool`. Both of these data structures have arrays of pointers to blocks in their `pool` fields, which are filled from index zero upwards. Thus, if `globalmem.pool[3]` is `NULL`, then the remainder of the array from index 4 up must also be `NULL`. The `cur` fields contain the index of the highest-numbered full element of the `pool` array, or -1 if all elements are empty. All elements from `globalmem.pool[0]` through `globalmem.pool[globalmem.cur]` must be full, and all the rest must be empty.[15]

The operation of the pool data structures is illustrated by Figure 6.20, with the six boxes representing the array of pointers making up the `pool` field, and the number

[15] Both pool sizes (`TARGET_POOL_SIZE` and `GLOBAL_POOL_SIZE`) are unrealistically small, but this small size makes it easier to single-step the program in order to get a feel for its operation.

www.dbooks.org

Listing 6.9: Allocator-Cache Data Structures

```
1 #define TARGET_POOL_SIZE 3
2 #define GLOBAL_POOL_SIZE 40
3
4 struct globalmempool {
5   spinlock_t mutex;
6   int cur;
7   struct memblock *pool[GLOBAL_POOL_SIZE];
8 } globalmem;
9
10 struct perthreadmempool {
11   int cur;
12   struct memblock *pool[2 * TARGET_POOL_SIZE];
13 };
14
15 DEFINE_PER_THREAD(struct perthreadmempool, perthreadmem);
```

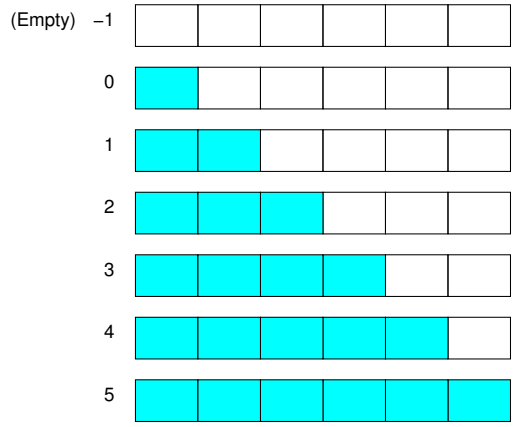


Figure 6.20: Allocator Pool Schematic

preceding them representing the `cur` field. The shaded boxes represent non-`NULL` pointers, while the empty boxes represent `NULL` pointers. An important, though potentially confusing, invariant of this data structure is that the `cur` field is always one smaller than the number of non-`NULL` pointers.

6.4.3.4 Allocation Function

The allocation function `memblock_alloc()` may be seen in Listing 6.10. Line 7 picks up the current thread's per-thread pool, and line 8 checks to see if it is empty.

If so, lines 9–16 attempt to refill it from the global pool under the spinlock acquired on line 9 and released on line 16. Lines 10–14 move blocks from the global to the per-thread pool until either the local pool reaches its target size (half full) or the global pool is exhausted, and line 15 sets the per-thread pool's count to the proper value.

In either case, line 18 checks for the per-thread pool still being empty, and if not, lines 19–21 remove a block and return it. Otherwise, line 23 tells the sad tale of memory exhaustion.

Listing 6.10: Allocator-Cache Allocator Function

```
1 struct memblock *memblock_alloc(void)
2 {
3   int i;
4   struct memblock *p;
5   struct perthreadmempool *pcpp;
6
7   pcpp = &__get_thread_var(perthreadmem);
8   if (pcpp->cur < 0) {
9     spin_lock(&globalmem.mutex);
10    for (i = 0; i < TARGET_POOL_SIZE &&
11                globalmem.cur >= 0; i++) {
12      pcpp->pool[i] = globalmem.pool[globalmem.cur];
13      globalmem.pool[globalmem.cur--] = NULL;
14    }
15    pcpp->cur = i - 1;
16    spin_unlock(&globalmem.mutex);
17  }
18  if (pcpp->cur >= 0) {
19    p = pcpp->pool[pcpp->cur];
20    pcpp->pool[pcpp->cur--] = NULL;
21    return p;
22  }
23  return NULL;
24 }
```

Listing 6.11: Allocator-Cache Free Function

```
1 void memblock_free(struct memblock *p)
2 {
3   int i;
4   struct perthreadmempool *pcpp;
5
6   pcpp = &__get_thread_var(perthreadmem);
7   if (pcpp->cur >= 2 * TARGET_POOL_SIZE - 1) {
8     spin_lock(&globalmem.mutex);
9     for (i = pcpp->cur; i >= TARGET_POOL_SIZE; i--) {
10      globalmem.pool[++globalmem.cur] = pcpp->pool[i];
11      pcpp->pool[i] = NULL;
12    }
13    pcpp->cur = i;
14    spin_unlock(&globalmem.mutex);
15  }
16  pcpp->pool[++pcpp->cur] = p;
17 }
```

6.4.3.5 Free Function

Listing 6.11 shows the memory-block free function. Line 6 gets a pointer to this thread's pool, and line 7 checks to see if this per-thread pool is full.

If so, lines 8–15 empty half of the per-thread pool into the global pool, with lines 8 and 14 acquiring and releasing the spinlock. Lines 9–12 implement the loop moving blocks from the local to the global pool, and line 13 sets the per-thread pool's count to the proper value.

In either case, line 16 then places the newly freed block into the per-thread pool.

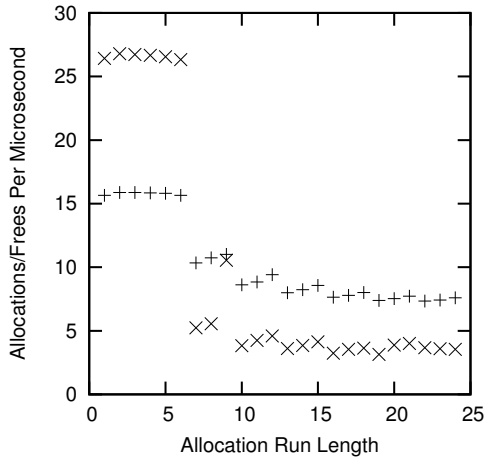


Figure 6.21: Allocator Cache Performance

Quick Quiz 6.18: Doesn't this resource-allocator design resemble that of the approximate limit counters covered in Section 5.3? ■

6.4.3.6 Performance

Rough performance results[16] are shown in Figure 6.21, running on a dual-core Intel x86 running at 1 GHz (4300 bogomips per CPU) with at most six blocks allowed in each CPU's cache. In this micro-benchmark, each thread repeatedly allocates a group of blocks and then frees all the blocks in that group, with the number of blocks in the group being the "allocation run length" displayed on the x-axis. The y-axis shows the number of successful allocation/free pairs per microsecond—failed allocations are not counted. The "X"s are from a two-thread run, while the "+"s are from a single-threaded run.

Note that run lengths up to six scale linearly and give excellent performance, while run lengths greater than six show poor performance and almost always also show *negative* scaling. It is therefore quite important to size `TARGET_POOL_SIZE` sufficiently large, which fortunately is usually quite easy to do in actual practice [MSK01], especially given today's large memories. For example, in most systems, it is quite reasonable to set `TARGET_POOL_SIZE` to 100, in which case allocations and frees are guaranteed to be confined to per-thread pools at least 99 % of the time.

As can be seen from the figure, the situations where the common-case data-ownership applies (run lengths up to six) provide greatly improved performance compared to the cases where locks must be acquired. Avoiding synchronization in the common case will be a recurring theme through this book.

Quick Quiz 6.19: In Figure 6.21, there is a pattern of performance rising with increasing run length in groups of three samples, for example, for run lengths 10, 11, and 12. Why? ■

Quick Quiz 6.20: Allocation failures were observed in the two-thread tests at run lengths of 19 and greater. Given the global-pool size of 40 and the per-thread target pool size s of three, number of threads n equal to two, and assuming that the per-thread pools are initially empty with none of the memory in use, what is the smallest allocation run length m at which failures can occur? (Recall that each thread repeatedly allocates m block of memory, and then frees the m blocks of memory.) Alternatively, given n threads each with pool size s, and where each thread repeatedly first allocates m blocks of memory and then frees those m blocks, how large must the global pool size be? *Note:* Obtaining the correct answer will require you to examine the `smpalloc.c` source code, and very likely single-step it as well. You have been warned! ■

6.4.3.7 Real-World Design

The toy parallel resource allocator was quite simple, but real-world designs expand on this approach in a number of ways.

First, real-world allocators are required to handle a wide range of allocation sizes, as opposed to the single size shown in this toy example. One popular way to do this is to offer a fixed set of sizes, spaced so as to balance external and internal fragmentation, such as in the late-1980s BSD memory allocator [MK88]. Doing this would mean that the "globalmem" variable would need to be replicated on a per-size basis, and that the associated lock would similarly be replicated, resulting in data locking rather than the toy program's code locking.

Second, production-quality systems must be able to repurpose memory, meaning that they must be able to coalesce blocks into larger structures, such as pages [MS93]. This coalescing will also need to be protected by a lock, which again could be replicated on a per-size basis.

Third, coalesced memory must be returned to the underlying memory system, and pages of memory must

[16] This data was not collected in a statistically meaningful way, and therefore should be viewed with great skepticism and suspicion. Good data-collection and -reduction practice is discussed in Chapter 11. That said, repeated runs gave similar results, and these results match more careful evaluations of similar algorithms.

www.dbooks.org

Table 6.1: Schematic of Real-World Parallel Allocator

Level	Locking	Purpose
Per-thread pool	Data ownership	High-speed allocation
Global block pool	Data locking	Distributing blocks among threads
Coalescing	Data locking	Combining blocks into pages
System memory	Code locking	Memory from/to system

also be allocated from the underlying memory system. The locking required at this level will depend on that of the underlying memory system, but could well be code locking. Code locking can often be tolerated at this level, because this level is so infrequently reached in well-designed systems [MSK01].

Despite this real-world design's greater complexity, the underlying idea is the same—repeated application of parallel fastpath, as shown in Table 6.1.

6.5 Beyond Partitioning

> It is all right to aim high if you have plenty of ammunition.
>
> *Hawley R. Everhart*

This chapter has discussed how data partitioning can be used to design simple linearly scalable parallel programs. Section 6.3.4 hinted at the possibilities of data replication, which will be used to great effect in Section 9.5.

The main goal of applying partitioning and replication is to achieve linear speedups, in other words, to ensure that the total amount of work required does not increase significantly as the number of CPUs or threads increases. A problem that can be solved via partitioning and/or replication, resulting in linear speedups, is *embarrassingly parallel*. But can we do better?

To answer this question, let us examine the solution of labyrinths and mazes. Of course, labyrinths and mazes have been objects of fascination for millennia [Wik12], so it should come as no surprise that they are generated and solved using computers, including biological computers [Ada11], GPGPUs [Eri08], and even discrete hardware [KFC11]. Parallel solution of mazes is sometimes used as a class project in universities [ETH11, Uni10] and as a vehicle to demonstrate the benefits of parallel-programming frameworks [Fos10].

Listing 6.12: SEQ Pseudocode

```
1  int maze_solve(maze *mp, cell sc, cell ec)
2  {
3    cell c = sc;
4    cell n;
5    int vi = 0;
6
7    maze_try_visit_cell(mp, c, c, &n, 1);
8    for (;;) {
9      while (!maze_find_any_next_cell(mp, c, &n)) {
10       if (++vi >= mp->vi)
11         return 0;
12       c = mp->visited[vi].c;
13     }
14     do {
15       if (n == ec) {
16         return 1;
17       }
18       c = n;
19     } while (maze_find_any_next_cell(mp, c, &n));
20     c = mp->visited[vi].c;
21   }
22 }
```

Common advice is to use a parallel work-queue algorithm (PWQ) [ETH11, Fos10]. This section evaluates this advice by comparing PWQ against a sequential algorithm (SEQ) and also against an alternative parallel algorithm, in all cases solving randomly generated square mazes. Section 6.5.1 discusses PWQ, Section 6.5.2 discusses an alternative parallel algorithm, Section 6.5.3 analyzes its anomalous performance, Section 6.5.4 derives an improved sequential algorithm from the alternative parallel algorithm, Section 6.5.5 makes further performance comparisons, and finally Section 6.5.6 presents future directions and concluding remarks.

6.5.1 Work-Queue Parallel Maze Solver

PWQ is based on SEQ, which is shown in Listing 6.12 (pseudocode for `maze_seq.c`). The maze is represented by a 2D array of cells and a linear-array-based work queue named `->visited`.

Line 7 visits the initial cell, and each iteration of the loop spanning lines 8–21 traverses passages headed by one cell. The loop spanning lines 9–13 scans the `->visited[]` array for a visited cell with an unvisited neighbor, and the loop spanning lines 14–19 traverses one fork of the submaze headed by that neighbor. Line 20 initializes for the next pass through the outer loop.

The pseudocode for `maze_try_visit_cell()` is shown on lines 1–12 of Listing 6.13 (`maze.c`). Line 4 checks to see if cells `c` and `t` are adjacent and connected,

Listing 6.13: SEQ Helper Pseudocode

```
1  int maze_try_visit_cell(struct maze *mp, cell c, cell t,
2                          cell *n, int d)
3  {
4    if (!maze_cells_connected(mp, c, t) ||
5        (*celladdr(mp, t) & VISITED))
6      return 0;
7    *n = t;
8    mp->visited[mp->vi] = t;
9    mp->vi++;
10   *celladdr(mp, t) |= VISITED | d;
11   return 1;
12 }
13
14 int maze_find_any_next_cell(struct maze *mp, cell c,
15                             cell *n)
16 {
17   int d = (*celladdr(mp, c) & DISTANCE) + 1;
18
19   if (maze_try_visit_cell(mp, c, prevcol(c), n, d))
20     return 1;
21   if (maze_try_visit_cell(mp, c, nextcol(c), n, d))
22     return 1;
23   if (maze_try_visit_cell(mp, c, prevrow(c), n, d))
24     return 1;
25   if (maze_try_visit_cell(mp, c, nextrow(c), n, d))
26     return 1;
27   return 0;
28 }
```

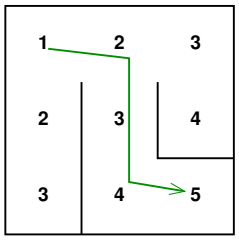


Figure 6.22: Cell-Number Solution Tracking

while line 5 checks to see if cell `t` has not yet been visited. The `celladdr()` function returns the address of the specified cell. If either check fails, line 6 returns failure. Line 7 indicates the next cell, line 8 records this cell in the next slot of the `->visited[]` array, line 9 indicates that this slot is now full, and line 10 marks this cell as visited and also records the distance from the maze start. Line 11 then returns success.

The pseudocode for `maze_find_any_next_cell()` is shown on lines 14–28 of Listing 6.13 (`maze.c`). Line 17 picks up the current cell's distance plus 1, while lines 19, 21, 23, and 25 check the cell in each direction, and lines 20, 22, 24, and 26 return true if the corresponding cell is a candidate next cell. The `prevcol()`, `nextcol()`, `prevrow()`, and `nextrow()` each do the specified array-index-conversion operation. If none of the cells is a candidate, line 27 returns false.

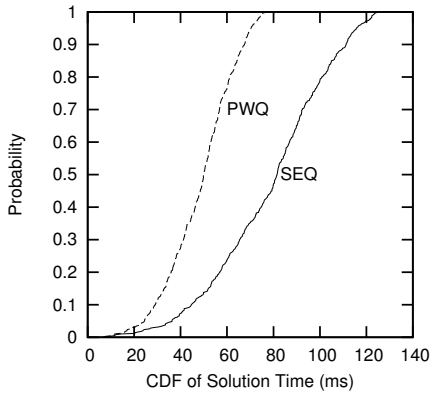


Figure 6.23: CDF of Solution Times For SEQ and PWQ

The path is recorded in the maze by counting the number of cells from the starting point, as shown in Figure 6.22, where the starting cell is in the upper left and the ending cell is in the lower right. Starting at the ending cell and following consecutively decreasing cell numbers traverses the solution.

The parallel work-queue solver is a straightforward parallelization of the algorithm shown in Listings 6.12 and 6.13. Line 10 of Listing 6.12 must use fetch-and-add, and the local variable `vi` must be shared among the various threads. Lines 5 and 10 of Listing 6.13 must be combined into a CAS loop, with CAS failure indicating a loop in the maze. Lines 8–9 of this listing must use fetch-and-add to arbitrate concurrent attempts to record cells in the `->visited[]` array.

This approach does provide significant speedups on a dual-CPU Lenovo W500 running at 2.53 GHz, as shown in Figure 6.23, which shows the cumulative distribution functions (CDFs) for the solution times of the two algorithms, based on the solution of 500 different square 500-by-500 randomly generated mazes. The substantial overlap of the projection of the CDFs onto the x-axis will be addressed in Section 6.5.3.

Interestingly enough, the sequential solution-path tracking works unchanged for the parallel algorithm. However, this uncovers a significant weakness in the parallel algorithm: At most one thread may be making progress along the solution path at any given time. This weakness is addressed in the next section.

6.5.2 Alternative Parallel Maze Solver

Youthful maze solvers are often urged to start at both ends, and this advice has been repeated more recently in the

www.dbooks.org

Listing 6.14: Partitioned Parallel Solver Pseudocode

```
1  int maze_solve_child(maze *mp, cell *visited, cell sc)
2  {
3    cell c;
4    cell n;
5    int vi = 0;
6
7    myvisited = visited; myvi = &vi;
8    c = visited[vi];
9    do {
10     while (!maze_find_any_next_cell(mp, c, &n)) {
11       if (visited[++vi].row < 0)
12         return 0;
13       if (READ_ONCE(mp->done))
14         return 1;
15       c = visited[vi];
16     }
17     do {
18       if (READ_ONCE(mp->done))
19         return 1;
20       c = n;
21     } while (maze_find_any_next_cell(mp, c, &n));
22     c = visited[vi];
23   } while (!READ_ONCE(mp->done));
24   return 1;
25 }
```

Listing 6.15: Partitioned Parallel Helper Pseudocode

```
1  int maze_try_visit_cell(struct maze *mp, int c, int t,
2                          int *n, int d)
3  {
4    cell_t t;
5    cell_t *tp;
6    int vi;
7
8    if (!maze_cells_connected(mp, c, t))
9      return 0;
10   tp = celladdr(mp, t);
11   do {
12     t = READ_ONCE(*tp);
13     if (t & VISITED) {
14       if ((t & TID) != mytid)
15         mp->done = 1;
16       return 0;
17     }
18   } while (!CAS(tp, t, t | VISITED | myid | d));
19   *n = t;
20   vi = (*myvi)++;
21   myvisited[vi] = t;
22   return 1;
23 }
```

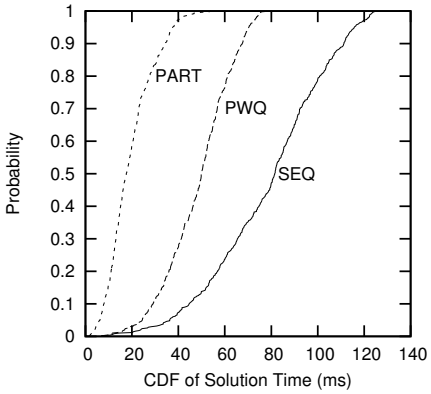


Figure 6.24: CDF of Solution Times For SEQ, PWQ, and PART

context of automated maze solving [Uni10]. This advice amounts to partitioning, which has been a powerful parallelization strategy in the context of parallel programming for both operating-system kernels [BK85, Inm85] and applications [Pat10]. This section applies this strategy, using two child threads that start at opposite ends of the solution path, and takes a brief look at the performance and scalability consequences.

The partitioned parallel algorithm (PART), shown in Listing 6.14 (`maze_part.c`), is similar to SEQ, but has a few important differences. First, each child thread has its own `visited` array, passed in by the parent as shown on line 1, which must be initialized to all [−1, −1]. Line 7 stores a pointer to this array into the per-thread variable `myvisited` to allow access by helper functions, and similarly stores a pointer to the local visit index. Second, the parent visits the first cell on each child's behalf, which the child retrieves on line 8. Third, the maze is solved as soon as one child locates a cell that has been visited by the other child. When `maze_try_visit_cell()` detects this, it sets a `->done` field in the maze structure. Fourth, each child must therefore periodically check the `->done` field, as shown on lines 13, 18, and 23. The `READ_ONCE()` primitive must disable any compiler optimizations that might combine consecutive loads or that might reload the value. A C++1x volatile relaxed load suffices [Smi19]. Finally, the `maze_find_any_next_cell()` function must use compare-and-swap to mark a cell as visited, however no constraints on ordering are required beyond those provided by thread creation and join.

The pseudocode for `maze_find_any_next_cell()` is identical to that shown in Listing 6.13, but the pseudocode for `maze_try_visit_cell()` differs, and is shown in Listing 6.15. Lines 8–9 check to see if the cells are connected, returning failure if not. The loop spanning lines 11–18 attempts to mark the new cell visited. Line 13 checks to see if it has already been visited, in which case line 16 returns failure, but only after line 14 checks to see if we have encountered the other thread, in which case line 15 indicates that the solution has been located. Line 19 updates to the new cell, lines 20 and 21 update this thread's visited array, and line 22 returns success.

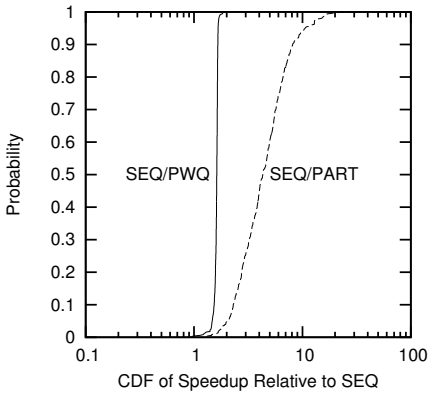


Figure 6.25: CDF of SEQ/PWQ and SEQ/PART Solution-Time Ratios

Performance testing revealed a surprising anomaly, shown in Figure 6.24. The median solution time for PART (17 milliseconds) is more than four times faster than that of SEQ (79 milliseconds), despite running on only two threads. The next section analyzes this anomaly.

6.5.3 Performance Comparison I

The first reaction to a performance anomaly is to check for bugs. Although the algorithms were in fact finding valid solutions, the plot of CDFs in Figure 6.24 assumes independent data points. This is not the case: The performance tests randomly generate a maze, and then run all solvers on that maze. It therefore makes sense to plot the CDF of the ratios of solution times for each generated maze, as shown in Figure 6.25, greatly reducing the CDFs' overlap. This plot reveals that for some mazes, PART is more than *forty* times faster than SEQ. In contrast, PWQ is never more than about two times faster than SEQ. A forty-times speedup on two threads demands explanation. After all, this is not merely embarrassingly parallel, where partitionability means that adding threads does not increase the overall computational cost. It is instead *humiliatingly parallel*: Adding threads significantly reduces the overall computational cost, resulting in large algorithmic superlinear speedups.

Further investigation showed that PART sometimes visited fewer than 2 % of the maze's cells, while SEQ and PWQ never visited fewer than about 9 %. The reason for this difference is shown by Figure 6.26. If the thread traversing the solution from the upper left reaches the circle, the other thread cannot reach the upper-right portion of the maze. Similarly, if the other thread reaches the

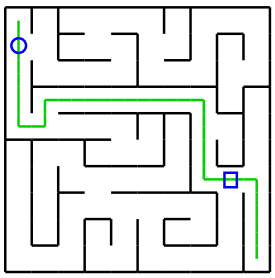


Figure 6.26: Reason for Small Visit Percentages

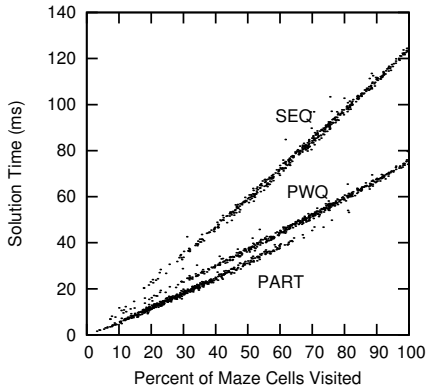


Figure 6.27: Correlation Between Visit Percentage and Solution Time

square, the first thread cannot reach the lower-left portion of the maze. Therefore, PART will likely visit a small fraction of the non-solution-path cells. In short, the superlinear speedups are due to threads getting in each others' way. This is a sharp contrast with decades of experience with parallel programming, where workers have struggled to keep threads *out* of each others' way.

Figure 6.27 confirms a strong correlation between cells visited and solution time for all three methods. The slope of PART's scatterplot is smaller than that of SEQ, indicating that PART's pair of threads visits a given fraction of the maze faster than can SEQ's single thread. PART's scatterplot is also weighted toward small visit percentages, confirming that PART does less total work, hence the observed humiliating parallelism.

The fraction of cells visited by PWQ is similar to that of SEQ. In addition, PWQ's solution time is greater than that of PART, even for equal visit fractions. The reason for this is shown in Figure 6.28, which has a red circle on each cell with more than two neighbors. Each such cell can result in contention in PWQ, because one thread can enter but two threads can exit, which hurts performance,

www.dbooks.org

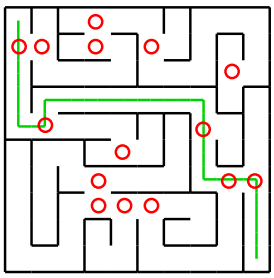


Figure 6.28: PWQ Potential Contention Points

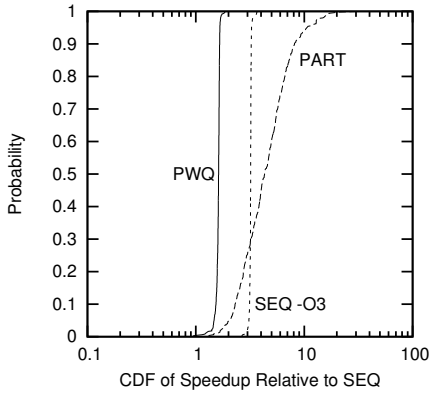


Figure 6.29: Effect of Compiler Optimization (-O3)

as noted earlier in this chapter. In contrast, PART can incur such contention but once, namely when the solution is located. Of course, SEQ never contends.

Although PART's speedup is impressive, we should not neglect sequential optimizations. Figure 6.29 shows that SEQ, when compiled with -O3, is about twice as fast as unoptimized PWQ, approaching the performance of unoptimized PART. Compiling all three algorithms with -O3 gives results similar to (albeit faster than) those shown in Figure 6.25, except that PWQ provides almost no speedup compared to SEQ, in keeping with Amdahl's Law [Amd67]. However, if the goal is to double performance compared to unoptimized SEQ, as opposed to achieving optimality, compiler optimizations are quite attractive.

Cache alignment and padding often improves performance by reducing false sharing. However, for these maze-solution algorithms, aligning and padding the maze-cell array *degrades* performance by up to 42 % for 1000x1000 mazes. Cache locality is more important than avoiding false sharing, especially for large mazes. For smaller 20-by-20 or 50-by-50 mazes, aligning and padding can produce up to a 40 % performance improvement for PART,

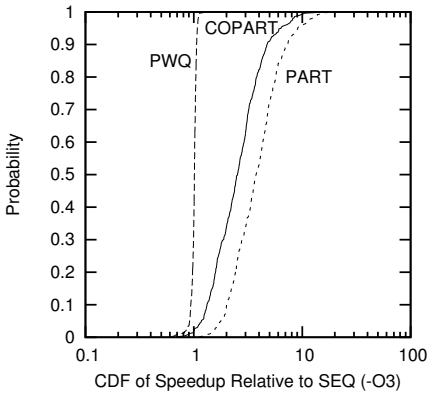


Figure 6.30: Partitioned Coroutines

but for these small sizes, SEQ performs better anyway because there is insufficient time for PART to make up for the overhead of thread creation and destruction.

In short, the partitioned parallel maze solver is an interesting example of an algorithmic superlinear speedup. If "algorithmic superlinear speedup" causes cognitive dissonance, please proceed to the next section.

6.5.4 Alternative Sequential Maze Solver

The presence of algorithmic superlinear speedups suggests simulating parallelism via co-routines, for example, manually switching context between threads on each pass through the main do-while loop in Listing 6.14. This context switching is straightforward because the context consists only of the variables `c` and `vi`: Of the numerous ways to achieve the effect, this is a good tradeoff between context-switch overhead and visit percentage. As can be seen in Figure 6.30, this coroutine algorithm (COPART) is quite effective, with the performance on one thread being within about 30 % of PART on two threads (`maze_2seq.c`).

6.5.5 Performance Comparison II

Figures 6.31 and 6.32 show the effects of varying maze size, comparing both PWQ and PART running on two threads against either SEQ or COPART, respectively, with 90-percent-confidence error bars. PART shows superlinear scalability against SEQ and modest scalability against COPART for 100-by-100 and larger mazes. PART exceeds theoretical energy-efficiency breakeven against COPART at roughly the 200-by-200 maze size, given that power consumption rises as roughly the square of the frequency

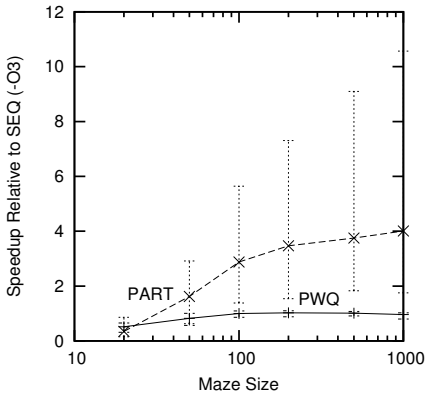


Figure 6.31: Varying Maze Size vs. SEQ

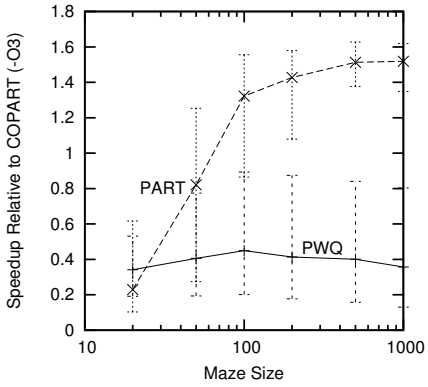


Figure 6.32: Varying Maze Size vs. COPART

for high frequencies [Mud01], so that 1.4x scaling on two threads consumes the same energy as a single thread at equal solution speeds. In contrast, PWQ shows poor scalability against both SEQ and COPART unless unoptimized: Figures 6.31 and 6.32 were generated using -O3.

Figure 6.33 shows the performance of PWQ and PART relative to COPART. For PART runs with more than two threads, the additional threads were started evenly spaced along the diagonal connecting the starting and ending cells. Simplified link-state routing [BG87] was used to detect early termination on PART runs with more than two threads (the solution is flagged when a thread is connected to both beginning and end). PWQ performs quite poorly, but PART hits breakeven at two threads and again at five threads, achieving modest speedups beyond five threads. Theoretical energy efficiency breakeven is within the 90-percent-confidence interval for seven and eight threads. The reasons for the peak at two threads are (1) the lower complexity of termination detection in

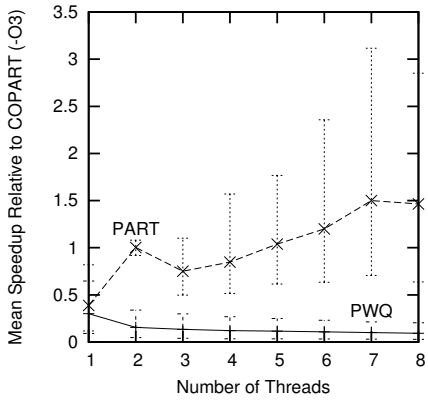


Figure 6.33: Mean Speedup vs. Number of Threads, 1000x1000 Maze

the two-thread case and (2) the fact that there is a lower probability of the third and subsequent threads making useful forward progress: Only the first two threads are guaranteed to start on the solution line. This disappointing performance compared to results in Figure 6.32 is due to the less-tightly integrated hardware available in the larger and older Xeon system running at 2.66 GHz.

6.5.6 Future Directions and Conclusions

Much future work remains. First, this section applied only one technique used by human maze solvers. Others include following walls to exclude portions of the maze and choosing internal starting points based on the locations of previously traversed paths. Second, different choices of starting and ending points might favor different algorithms. Third, although placement of the PART algorithm's first two threads is straightforward, there are any number of placement schemes for the remaining threads. Optimal placement might well depend on the starting and ending points. Fourth, study of unsolvable mazes and cyclic mazes is likely to produce interesting results. Fifth, the lightweight C++11 atomic operations might improve performance. Sixth, it would be interesting to compare the speedups for three-dimensional mazes (or of even higher-order mazes). Finally, for mazes, humiliating parallelism indicated a more-efficient sequential implementation using coroutines. Do humiliatingly parallel algorithms always lead to more-efficient sequential implementations, or are there inherently humiliatingly parallel algorithms for which coroutine context-switch overhead overwhelms the speedups?

www.dbooks.org

This section demonstrated and analyzed parallelization of maze-solution algorithms. A conventional work-queue-based algorithm did well only when compiler optimizations were disabled, suggesting that some prior results obtained using high-level/overhead languages will be invalidated by advances in optimization.

This section gave a clear example where approaching parallelism as a first-class optimization technique rather than as a derivative of a sequential algorithm paves the way for an improved sequential algorithm. High-level design-time application of parallelism is likely to be a fruitful field of study. This section took the problem of solving mazes from mildly scalable to humiliatingly parallel and back again. It is hoped that this experience will motivate work on parallelism as a first-class design-time whole-application optimization technique, rather than as a grossly suboptimal after-the-fact micro-optimization to be retrofitted into existing programs.

6.6 Partitioning, Parallelism, and Optimization

> Knowledge is of no value unless you put it into practice.
>
> *Anton Chekhov*

Most important, although this chapter has demonstrated that applying parallelism at the design level gives excellent results, this final section shows that this is not enough. For search problems such as maze solution, this section has shown that search strategy is even more important than parallel design. Yes, for this particular type of maze, intelligently applying parallelism identified a superior search strategy, but this sort of luck is no substitute for a clear focus on search strategy itself.

As noted back in Section 2.2, parallelism is but one potential optimization of many. A successful design needs to focus on the most important optimization. Much though I might wish to claim otherwise, that optimization might or might not be parallelism.

However, for the many cases where parallelism is the right optimization, the next section covers that synchronization workhorse, locking.

Chapter 7

Locking

> Locking is the worst general-purpose synchronization mechanism except for all those other mechanisms that have been tried from time to time.
>
> *With apologies to the memory of Winston Churchill and to whoever he was quoting*

In recent concurrency research, locking often plays the role of villain. Locking stands accused of inciting deadlocks, convoying, starvation, unfairness, data races, and all manner of other concurrency sins. Interestingly enough, the role of workhorse in production-quality shared-memory parallel software is also played by locking. This chapter will look into this dichotomy between villain and hero, as fancifully depicted in Figures 7.1 and 7.2.

There are a number of reasons behind this Jekyll-and-Hyde dichotomy:

1. Many of locking's sins have pragmatic design solutions that work well in most cases, for example:
   (a) Use of lock hierarchies to avoid deadlock.
   (b) Deadlock-detection tools, for example, the Linux kernel's lockdep facility [Cor06a].
   (c) Locking-friendly data structures, such as arrays, hash tables, and radix trees, which will be covered in Chapter 10.
2. Some of locking's sins are problems only at high levels of contention, levels reached only by poorly designed programs.
3. Some of locking's sins are avoided by using other synchronization mechanisms in concert with locking. These other mechanisms include statistical counters (see Chapter 5), reference counters (see Section 9.2), hazard pointers (see Section 9.3), sequence-locking readers (see Section 9.4), RCU (see Section 9.5), and simple non-blocking data structures (see Section 14.2).
4. Until quite recently, almost all large shared-memory parallel programs were developed in secret, so that it was not easy to learn of these pragmatic solutions.
5. Locking works extremely well for some software artifacts and extremely poorly for others. Developers who have worked on artifacts for which locking works well can be expected to have a much more positive opinion of locking than those who have worked on artifacts for which locking works poorly, as will be discussed in Section 7.5.
6. All good stories need a villain, and locking has a long and honorable history serving as a research-paper whipping boy.

Quick Quiz 7.1: Just how can serving as a whipping boy be considered to be in any way honorable??? ■

This chapter will give an overview of a number of ways to avoid locking's more serious sins.

7.1 Staying Alive

> I work to stay alive.
>
> *Bette Davis*

Given that locking stands accused of deadlock and starvation, one important concern for shared-memory parallel developers is simply staying alive. The following sections therefore cover deadlock, livelock, starvation, unfairness, and inefficiency.

7.1.1 Deadlock

Deadlock occurs when each member of a group of threads is holding at least one lock while at the same time waiting on a lock held by a member of that same group. This happens even in groups containing a single thread when

Edition.2-rc11

www.dbooks.org

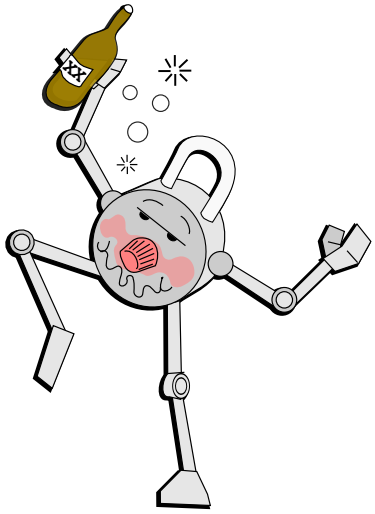


Figure 7.1: Locking: Villain or Slob?

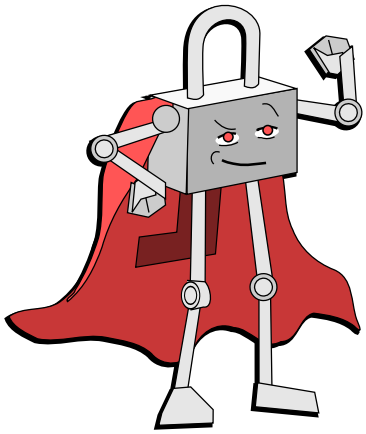


Figure 7.2: Locking: Workhorse or Hero?

that thread attempts to acquire a non-recursive lock that it already holds. Deadlock can therefore occur even given but one thread and one lock!

Without some sort of external intervention, deadlock is forever. No thread can acquire the lock it is waiting on until that lock is released by the thread holding it, but the thread holding it cannot release it until the holding thread acquires the lock that it is in turn waiting on.

We can create a directed-graph representation of a deadlock scenario with nodes for threads and locks, as shown in Figure 7.3. An arrow from a lock to a thread indicates that the thread holds the lock, for example, Thread B holds Locks 2 and 4. An arrow from a thread to a lock indicates that the thread is waiting on the lock, for example, Thread B is waiting on Lock 3.

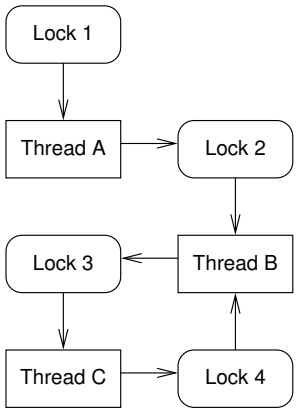


Figure 7.3: Deadlock Cycle

A deadlock scenario will always contain at least one deadlock cycle. In Figure 7.3, this cycle is Thread B, Lock 3, Thread C, Lock 4, and back to Thread B.

Quick Quiz 7.2: But the definition of lock-based deadlock only said that each thread was holding at least one lock and waiting on another lock that was held by some thread. How do you know that there is a cycle? ■

Although there are some software environments such as database systems that can recover from an existing deadlock, this approach requires either that one of the threads be killed or that a lock be forcibly stolen from one of the threads. This killing and forcible stealing works well for transactions, but is often problematic for kernel and application-level use of locking: dealing with the resulting partially updated structures can be extremely complex, hazardous, and error-prone.

Therefore, kernels and applications should instead avoid deadlocks. Deadlock-avoidance strategies include locking hierarchies (Section 7.1.1.1), local locking hierarchies (Section 7.1.1.2), layered locking hierarchies (Section 7.1.1.3), strategies for dealing with APIs containing pointers to locks (Section 7.1.1.4), conditional locking (Section 7.1.1.5), acquiring all needed locks first (Section 7.1.1.6), single-lock-at-a-time designs (Section 7.1.1.7), and strategies for signal/interrupt handlers (Section 7.1.1.8). Although there is no deadlock-avoidance strategy that works perfectly for all situations, there is a good selection of tools to choose from.

7.1.1.1 Locking Hierarchies

Locking hierarchies order the locks and prohibit acquiring locks out of order. In Figure 7.3, we might order the locks numerically, thus forbidding a thread from acquiring a given lock if it already holds a lock with the same or a higher number. Thread B has violated this hierarchy because it is attempting to acquire Lock 3 while holding Lock 4. This violation permitted the deadlock to occur.

Again, to apply a locking hierarchy, order the locks and prohibit out-of-order lock acquisition. In large program, it is wise to use tools such as the Linux-kernel `lockdep` [Cor06a] to enforce your locking hierarchy.

7.1.1.2 Local Locking Hierarchies

However, the global nature of locking hierarchies make them difficult to apply to library functions. After all, when a program using a given library function has not yet been written, how can the poor library-function implementor possibly follow the yet-to-be-defined locking hierarchy?

One special (but common) case is when the library function does not invoke any of the caller's code. In this case, the caller's locks will never be acquired while holding any of the library's locks, so that there cannot be a deadlock cycle containing locks from both the library and the caller.

Quick Quiz 7.3: Are there any exceptions to this rule, so that there really could be a deadlock cycle containing locks from both the library and the caller, even given that the library code never invokes any of the caller's functions? ■

But suppose that a library function does invoke the caller's code. For example, `qsort()` invokes a caller-provided comparison function. Now, normally this comparison function will operate on unchanging local data, so that it need not acquire locks, as shown in Figure 7.4. But maybe someone is crazy enough to sort a collection whose keys are changing, thus requiring that the comparison function acquire locks, which might result in deadlock, as shown in Figure 7.5. How can the library function avoid this deadlock?

The golden rule in this case is "Release all locks before invoking unknown code." To follow this rule, the `qsort()` function must release all of its locks before invoking the comparison function. Thus `qsort()` will not be holding any of its locks while the comparison function acquires any of the caller's locks, thus avoiding deadlock.

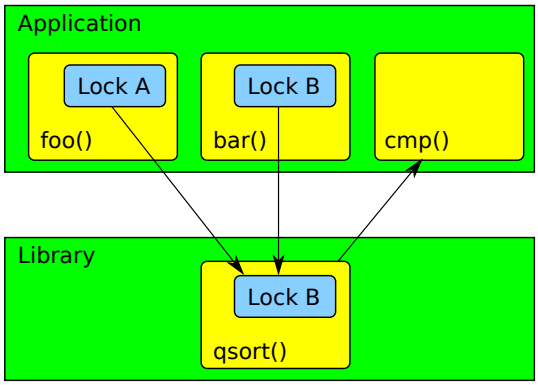


Figure 7.4: No `qsort()` Compare-Function Locking

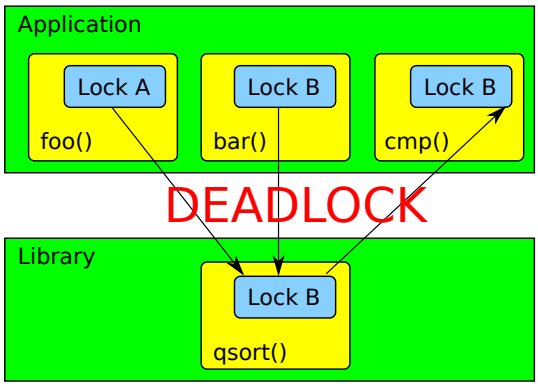


Figure 7.5: Without `qsort()` Local Locking Hierarchy

Quick Quiz 7.4: But if `qsort()` releases all its locks before invoking the comparison function, how can it protect against races with other `qsort()` threads? ■

To see the benefits of local locking hierarchies, compare Figures 7.5 and 7.6. In both figures, application functions `foo()` and `bar()` invoke `qsort()` while holding Locks A and B, respectively. Because this is a parallel implementation of `qsort()`, it acquires Lock C. Function `foo()` passes function `cmp()` to `qsort()`, and `cmp()` acquires Lock B. Function `bar()` passes a simple integer-comparison function (not shown) to `qsort()`, and this simple function does not acquire any locks.

Now, if `qsort()` holds Lock C while calling `cmp()` in violation of the golden release-all-locks rule above, as shown in Figure 7.5, deadlock can occur. To see this, suppose that one thread invokes `foo()` while a second

www.dbooks.org

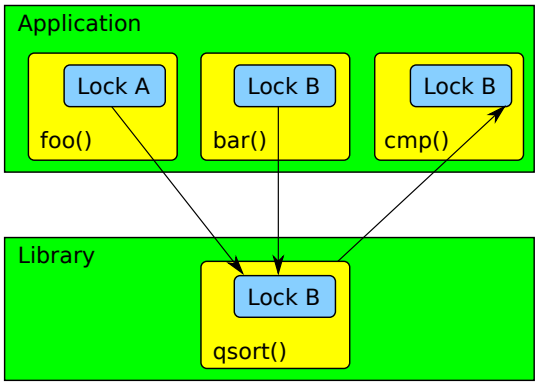


Figure 7.6: Local Locking Hierarchy for `qsort()`

thread concurrently invokes `bar()`. The first thread will acquire Lock A and the second thread will acquire Lock B. If the first thread's call to `qsort()` acquires Lock C, then it will be unable to acquire Lock B when it calls `cmp()`. But the first thread holds Lock C, so the second thread's call to `qsort()` will be unable to acquire it, and thus unable to release Lock B, resulting in deadlock.

In contrast, if `qsort()` releases Lock C before invoking the comparison function, which is unknown code from `qsort()`'s perspective, then deadlock is avoided as shown in Figure 7.6.

If each module releases all locks before invoking unknown code, then deadlock is avoided if each module separately avoids deadlock. This rule therefore greatly simplifies deadlock analysis and greatly improves modularity.

7.1.1.3 Layered Locking Hierarchies

Unfortunately, it might not be possible for `qsort()` to release all of its locks before invoking the comparison function. In this case, we cannot construct a local locking hierarchy by releasing all locks before invoking unknown code. However, we can instead construct a layered locking hierarchy, as shown in Figure 7.7. here, the `cmp()` function uses a new Lock D that is acquired after all of Locks A, B, and C, avoiding deadlock. We therefore have three layers to the global deadlock hierarchy, the first containing Locks A and B, the second containing Lock C, and the third containing Lock D.

Please note that it is not typically possible to mechanically change `cmp()` to use the new Lock D. Quite the opposite: It is often necessary to make profound design-

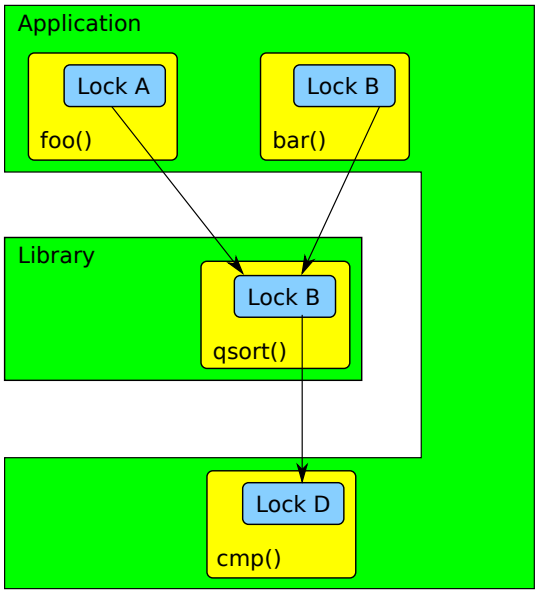


Figure 7.7: Layered Locking Hierarchy for `qsort()`

level modifications. Nevertheless, the effort required for such modifications is normally a small price to pay in order to avoid deadlock. More to the point, this potential deadlock should preferably be detected at design time, before any code has been generated!

For another example where releasing all locks before invoking unknown code is impractical, imagine an iterator over a linked list, as shown in Listing 7.1 (`locked_list.c`). The `list_start()` function acquires a lock on the list and returns the first element (if there is one), and `list_next()` either returns a pointer to the next element in the list or releases the lock and returns `NULL` if the end of the list has been reached.

Listing 7.2 shows how this list iterator may be used. Lines 1–4 define the `list_ints` element containing a single integer, and lines 6–17 show how to iterate over the list. Line 11 locks the list and fetches a pointer to the first element, line 13 provides a pointer to our enclosing `list_ints` structure, line 14 prints the corresponding integer, and line 15 moves to the next element. This is quite simple, and hides all of the locking.

That is, the locking remains hidden as long as the code processing each list element does not itself acquire a lock that is held across some other call to `list_start()` or `list_next()`, which results in deadlock. We can avoid

Listing 7.1: Concurrent List Iterator

```
1  struct locked_list {
2    spinlock_t s;
3    struct cds_list_head h;
4  };
5
6  struct cds_list_head *list_start(struct locked_list *lp)
7  {
8    spin_lock(&lp->s);
9    return list_next(lp, &lp->h);
10 }
11
12 struct cds_list_head *list_next(struct locked_list *lp,
13                                 struct cds_list_head *np)
14 {
15   struct cds_list_head *ret;
16
17   ret = np->next;
18   if (ret == &lp->h) {
19     spin_unlock(&lp->s);
20     ret = NULL;
21   }
22   return ret;
23 }
```

Listing 7.2: Concurrent List Iterator Usage

```
1  struct list_ints {
2    struct cds_list_head n;
3    int a;
4  };
5
6  void list_print(struct locked_list *lp)
7  {
8    struct cds_list_head *np;
9    struct list_ints *ip;
10
11   np = list_start(lp);
12   while (np != NULL) {
13     ip = cds_list_entry(np, struct list_ints, n);
14     printf("\t%d\n", ip->a);
15     np = list_next(lp, np);
16   }
17 }
```

the deadlock by layering the locking hierarchy to take the list-iterator locking into account.

This layered approach can be extended to an arbitrarily large number of layers, but each added layer increases the complexity of the locking design. Such increases in complexity are particularly inconvenient for some types of object-oriented designs, in which control passes back and forth among a large group of objects in an undisciplined manner.[1] This mismatch between the habits of object-oriented design and the need to avoid deadlock is an important reason why parallel programming is perceived by some to be so difficult.

Some alternatives to highly layered locking hierarchies are covered in Chapter 9.

7.1.1.4 Locking Hierarchies and Pointers to Locks

Although there are some exceptions, an external API containing a pointer to a lock is very often a misdesigned API. Handing an internal lock to some other software component is after all the antithesis of information hiding, which is in turn a key design principle.

Quick Quiz 7.5: Name one common situation where a pointer to a lock is passed into a function. ■

One exception is functions that hand off some entity, where the caller's lock must be held until the handoff is complete, but where the lock must be released before the function returns. One example of such a function is the POSIX `pthread_cond_wait()` function, where passing an pointer to a `pthread_mutex_t` prevents hangs due to lost wakeups.

Quick Quiz 7.6: Doesn't the fact that `pthread_cond_wait()` first releases the mutex and then re-acquires it eliminate the possibility of deadlock? ■

In short, if you find yourself exporting an API with a pointer to a lock as an argument or the as the return value, do yourself a favor and carefully reconsider your API design. It might well be the right thing to do, but experience indicates that this is unlikely.

7.1.1.5 Conditional Locking

But suppose that there is no reasonable locking hierarchy. This can happen in real life, for example, in some types of layered network protocol stacks where packets flow in both directions, for example, in implementations of distributed lock managers. In the networking case, it

[1] One name for this is "object-oriented spaghetti code."

www.dbooks.org

Listing 7.3: Protocol Layering and Deadlock

```
1 spin_lock(&lock2);
2 layer_2_processing(pkt);
3 nextlayer = layer_1(pkt);
4 spin_lock(&nextlayer->lock1);
5 layer_1_processing(pkt);
6 spin_unlock(&lock2);
7 spin_unlock(&nextlayer->lock1);
```

Listing 7.4: Avoiding Deadlock Via Conditional Locking

```
1  retry:
2    spin_lock(&lock2);
3    layer_2_processing(pkt);
4    nextlayer = layer_1(pkt);
5    if (!spin_trylock(&nextlayer->lock1)) {
6      spin_unlock(&lock2);
7      spin_lock(&nextlayer->lock1);
8      spin_lock(&lock2);
9      if (layer_1(pkt) != nextlayer) {
10       spin_unlock(&nextlayer->lock1);
11       spin_unlock(&lock2);
12       goto retry;
13     }
14   }
15   layer_1_processing(pkt);
16   spin_unlock(&lock2);
17   spin_unlock(&nextlayer->lock1);
```

might be necessary to hold the locks from both layers when passing a packet from one layer to another. Given that packets travel both up and down the protocol stack, this is an excellent recipe for deadlock, as illustrated in Listing 7.3. Here, a packet moving down the stack towards the wire must acquire the next layer's lock out of order. Given that packets moving up the stack away from the wire are acquiring the locks in order, the lock acquisition in line 4 of the listing can result in deadlock.

One way to avoid deadlocks in this case is to impose a locking hierarchy, but when it is necessary to acquire a lock out of order, acquire it conditionally, as shown in Listing 7.4. Instead of unconditionally acquiring the layer-1 lock, line 5 conditionally acquires the lock using the `spin_trylock()` primitive. This primitive acquires the lock immediately if the lock is available (returning non-zero), and otherwise returns zero without acquiring the lock.

If `spin_trylock()` was successful, line 15 does the needed layer-1 processing. Otherwise, line 6 releases the lock, and lines 7 and 8 acquire them in the correct order. Unfortunately, there might be multiple networking devices on the system (e.g., Ethernet and WiFi), so that the `layer_1()` function must make a routing decision. This decision might change at any time, especially if the system is mobile.[2] Therefore, line 9 must recheck the decision, and if it has changed, must release the locks and start over.

[2] And, in contrast to the 1900s, mobility is the common case.

Quick Quiz 7.7: Can the transformation from Listing 7.3 to Listing 7.4 be applied universally? ■

Quick Quiz 7.8: But the complexity in Listing 7.4 is well worthwhile given that it avoids deadlock, right? ■

7.1.1.6 Acquire Needed Locks First

In an important special case of conditional locking, all needed locks are acquired before any processing is carried out. In this case, processing need not be idempotent: if it turns out to be impossible to acquire a given lock without first releasing one that was already acquired, just release all the locks and try again. Only once all needed locks are held will any processing be carried out.

However, this procedure can result in *livelock*, which will be discussed in Section 7.1.2.

Quick Quiz 7.9: When using the "acquire needed locks first" approach described in Section 7.1.1.6, how can livelock be avoided? ■

A related approach, two-phase locking [BHG87], has seen long production use in transactional database systems. In the first phase of a two-phase locking transaction, locks are acquired but not released. Once all needed locks have been acquired, the transaction enters the second phase, where locks are released, but not acquired. This locking approach allows databases to provide serializability guarantees for their transactions, in other words, to guarantee that all values seen and produced by the transactions are consistent with some global ordering of all the transactions. Many such systems rely on the ability to abort transactions, although this can be simplified by avoiding making any changes to shared data until all needed locks are acquired. Livelock and deadlock are issues in such systems, but practical solutions may be found in any of a number of database textbooks.

7.1.1.7 Single-Lock-at-a-Time Designs

In some cases, it is possible to avoid nesting locks, thus avoiding deadlock. For example, if a problem is perfectly partitionable, a single lock may be assigned to each partition. Then a thread working on a given partition need only acquire the one corresponding lock. Because no thread ever holds more than one lock at a time, deadlock is impossible.

However, there must be some mechanism to ensure that the needed data structures remain in existence during the time that neither lock is held. One such mechanism is discussed in Section 7.4 and several others are presented in Chapter 9.

7.1.1.8 Signal/Interrupt Handlers

Deadlocks involving signal handlers are often quickly dismissed by noting that it is not legal to invoke `pthread_mutex_lock()` from within a signal handler [Ope97]. However, it is possible (though often unwise) to hand-craft locking primitives that *can* be invoked from signal handlers. Besides which, almost all operating-system kernels permit locks to be acquired from within interrupt handlers, which are analogous to signal handlers.

The trick is to block signals (or disable interrupts, as the case may be) when acquiring any lock that might be acquired within a signal (or an interrupt) handler. Furthermore, if holding such a lock, it is illegal to attempt to acquire any lock that is ever acquired outside of a signal handler without blocking signals.

Quick Quiz 7.10: Suppose Lock A is never acquired within a signal handler, but Lock B is acquired both from thread context and by signal handlers. Suppose further that Lock A is sometimes acquired with signals unblocked. Why is it illegal to acquire Lock A holding Lock B? ■

If a lock is acquired by the handlers for several signals, then each and every one of these signals must be blocked whenever that lock is acquired, even when that lock is acquired within a signal handler.

Quick Quiz 7.11: How can you legally block signals within a signal handler? ■

Unfortunately, blocking and unblocking signals can be expensive in some operating systems, notably including Linux, so performance concerns often mean that locks acquired in signal handlers are only acquired in signal handlers, and that lockless synchronization mechanisms are used to communicate between application code and signal handlers.

Or that signal handlers are avoided completely except for handling fatal errors.

Quick Quiz 7.12: If acquiring locks in signal handlers is such a bad idea, why even discuss ways of making it safe? ■

Listing 7.5: Abusing Conditional Locking

```
1  void thread1(void)
2  {
3  retry:
4    spin_lock(&lock1);
5    do_one_thing();
6    if (!spin_trylock(&lock2)) {
7      spin_unlock(&lock1);
8      goto retry;
9    }
10   do_another_thing();
11   spin_unlock(&lock2);
12   spin_unlock(&lock1);
13 }
14
15 void thread2(void)
16 {
17 retry:
18   spin_lock(&lock2);
19   do_a_third_thing();
20   if (!spin_trylock(&lock1)) {
21     spin_unlock(&lock2);
22     goto retry;
23   }
24   do_a_fourth_thing();
25   spin_unlock(&lock1);
26   spin_unlock(&lock2);
27 }
```

7.1.1.9 Discussion

There are a large number of deadlock-avoidance strategies available to the shared-memory parallel programmer, but there are sequential programs for which none of them is a good fit. This is one of the reasons that expert programmers have more than one tool in their toolbox: locking is a powerful concurrency tool, but there are jobs better addressed with other tools.

Quick Quiz 7.13: Given an object-oriented application that passes control freely among a group of objects such that there is no straightforward locking hierarchy,[a] layered or otherwise, how can this application be parallelized? ■

[a] Also known as "object-oriented spaghetti code."

Nevertheless, the strategies described in this section have proven quite useful in many settings.

7.1.2 Livelock and Starvation

Although conditional locking can be an effective deadlock-avoidance mechanism, it can be abused. Consider for example the beautifully symmetric example shown in Listing 7.5. This example's beauty hides an ugly livelock. To see this, consider the following sequence of events:

1. Thread 1 acquires `lock1` on line 4, then invokes `do_one_thing()`.

www.dbooks.org

2. Thread 2 acquires `lock2` on line 18, then invokes `do_a_third_thing()`.
3. Thread 1 attempts to acquire `lock2` on line 6, but fails because Thread 2 holds it.
4. Thread 2 attempts to acquire `lock1` on line 20, but fails because Thread 1 holds it.
5. Thread 1 releases `lock1` on line 7, then jumps to `retry` at line 3.
6. Thread 2 releases `lock2` on line 21, and jumps to `retry` at line 17.
7. The livelock dance repeats from the beginning.

Quick Quiz 7.14: How can the livelock shown in Listing 7.5 be avoided? ■

Livelock can be thought of as an extreme form of starvation where a group of threads starves, rather than just one of them.[3]

Livelock and starvation are serious issues in software transactional memory implementations, and so the concept of *contention manager* has been introduced to encapsulate these issues. In the case of locking, simple exponential backoff can often address livelock and starvation. The idea is to introduce exponentially increasing delays before each retry, as shown in Listing 7.6.

Quick Quiz 7.15: What problems can you spot in the code in Listing 7.6? ■

For better results, backoffs should be bounded, and even better high-contention results are obtained via queued locking [And90], which is discussed more in Section 7.3.2. Of course, best of all is to use a good parallel design that avoids these problems by maintaining low lock contention.

7.1.3 Unfairness

Unfairness can be thought of as a less-severe form of starvation, where a subset of threads contending for a given lock are granted the lion's share of the acquisitions. This can happen on machines with shared caches or NUMA characteristics, for example, as shown in Figure 7.8. If CPU 0 releases a lock that all the other CPUs are attempting to acquire, the interconnect shared between CPUs 0 and 1 means that CPU 1 will have an advantage over CPUs 2–7. Therefore CPU 1 will likely acquire the lock. If CPU 1 holds the lock long enough for CPU 0 to be requesting the lock by the time CPU 1 releases it and vice versa, the lock can shuttle between CPUs 0 and 1, bypassing CPUs 2–7.

[3] Try not to get too hung up on the exact definitions of terms like livelock, starvation, and unfairness. Anything that causes a group of threads to fail to make adequate forward progress is a bug that needs to be fixed, and debating names doesn't fix bugs.

Listing 7.6: Conditional Locking and Exponential Backoff

```
1  void thread1(void)
2  {
3    unsigned int wait = 1;
4  retry:
5    spin_lock(&lock1);
6    do_one_thing();
7    if (!spin_trylock(&lock2)) {
8      spin_unlock(&lock1);
9      sleep(wait);
10     wait = wait << 1;
11     goto retry;
12   }
13   do_another_thing();
14   spin_unlock(&lock2);
15   spin_unlock(&lock1);
16 }
17
18 void thread2(void)
19 {
20   unsigned int wait = 1;
21 retry:
22   spin_lock(&lock2);
23   do_a_third_thing();
24   if (!spin_trylock(&lock1)) {
25     spin_unlock(&lock2);
26     sleep(wait);
27     wait = wait << 1;
28     goto retry;
29   }
30   do_a_fourth_thing();
31   spin_unlock(&lock1);
32   spin_unlock(&lock2);
33 }
```

Quick Quiz 7.16: Wouldn't it be better just to use a good parallel design so that lock contention was low enough to avoid unfairness? ■

7.1.4 Inefficiency

Locks are implemented using atomic instructions and memory barriers, and often involve cache misses. As we saw in Chapter 3, these instructions are quite expensive, roughly two orders of magnitude greater overhead than simple instructions. This can be a serious problem for locking: If you protect a single instruction with a lock, you will increase the overhead by a factor of one hundred. Even assuming perfect scalability, *one hundred* CPUs would be required to keep up with a single CPU executing the same code without locking.

This situation underscores the synchronization-granularity tradeoff discussed in Section 6.3, especially Figure 6.16:

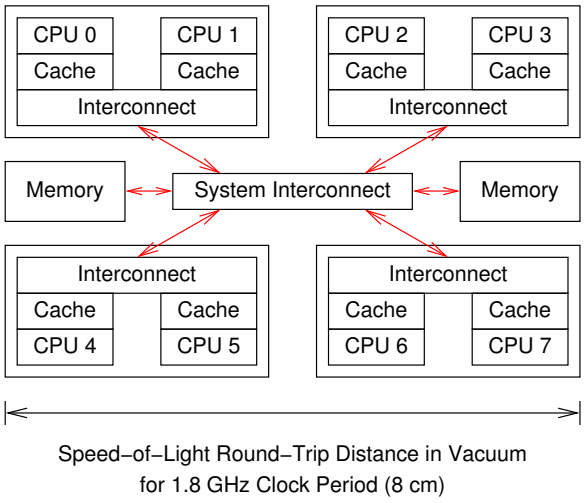


Figure 7.8: System Architecture and Lock Unfairness

Too coarse a granularity will limit scalability, while too fine a granularity will result in excessive synchronization overhead.

Acquiring a lock might be expensive, but once held, the CPU's caches are an effective performance booster, at least for large critical sections. In addition, once a lock is held, the data protected by that lock can be accessed by the lock holder without interference from other threads.

Quick Quiz 7.17: How might the lock holder be interfered with? ■

7.2 Types of Locks

> Only locks in life are what you think you know, but don't. Accept your ignorance and try something new.
>
> *Dennis Vickers*

There are a surprising number of types of locks, more than this short chapter can possibly do justice to. The following sections discuss exclusive locks (Section 7.2.1), reader-writer locks (Section 7.2.2), multi-role locks (Section 7.2.3), and scoped locking (Section 7.2.4).

7.2.1 Exclusive Locks

Exclusive locks are what they say they are: only one thread may hold the lock at a time. The holder of such a lock thus has exclusive access to all data protected by that lock, hence the name.

Of course, this all assumes that this lock is held across all accesses to data purportedly protected by the lock. Although there are some tools that can help (see for example Section 12.3.1), the ultimate responsibility for ensuring that the lock is always acquired when needed rests with the developer.

Quick Quiz 7.18: Does it ever make sense to have an exclusive lock acquisition immediately followed by a release of that same lock, that is, an empty critical section? ■

It is important to note that unconditionally acquiring an exclusive lock has two effects: (1) Waiting for all prior holders of that lock to release it, and (2) Blocking any other acquisition attempts until the lock is released. As a result, at lock acquisition time, any concurrent acquisitions of that lock must be partitioned into prior holders and subsequent holders. Different types of exclusive locks use different partitioning strategies [Bra11, GGL+19], for example:

1. Strict FIFO, with acquisitions starting earlier acquiring the lock earlier.

2. Approximate FIFO, with acquisitions starting sufficiently earlier acquiring the lock earlier.

3. FIFO within priority level, with higher-priority threads acquiring the lock earlier than any lower-priority threads attempting to acquire the lock at about the same time, but so that some FIFO ordering applies for threads of the same priority.

4. Random, so that the new lock holder is chosen randomly from all threads attempting acquisition, regardless of timing.

5. Unfair, so that a given acquisition might never acquire the lock (see Section 7.1.3).

Unfortunately, locking implementations with stronger guarantees typically incur higher overhead, motivating the wide variety of locking implementations in production use. For example, real-time systems often require some degree of FIFO ordering within priority level, and much else besides (see Section 14.3.5.1), while non-realtime systems subject to high contention might require only enough ordering to avoid starvation, and finally, non-realtime systems designed to avoid contention might not need fairness at all.

www.dbooks.org

7.2.2 Reader-Writer Locks

Reader-writer locks [CHP71] permit any number of readers to hold the lock concurrently on the one hand or a single writer to hold the lock on the other. In theory, then, reader-writer locks should allow excellent scalability for data that is read often and written rarely. In practice, the scalability will depend on the reader-writer lock implementation.

The classic reader-writer lock implementation involves a set of counters and flags that are manipulated atomically. This type of implementation suffers from the same problem as does exclusive locking for short critical sections: The overhead of acquiring and releasing the lock is about two orders of magnitude greater than the overhead of a simple instruction. Of course, if the critical section is long enough, the overhead of acquiring and releasing the lock becomes negligible. However, because only one thread at a time can be manipulating the lock, the required critical-section size increases with the number of CPUs.

It is possible to design a reader-writer lock that is much more favorable to readers through use of per-thread exclusive locks [HW92]. To read, a thread acquires only its own lock. To write, a thread acquires all locks. In the absence of writers, each reader incurs only atomic-instruction and memory-barrier overhead, with no cache misses, which is quite good for a locking primitive. Unfortunately, writers must incur cache misses as well as atomic-instruction and memory-barrier overhead—multiplied by the number of threads.

In short, reader-writer locks can be quite useful in a number of situations, but each type of implementation does have its drawbacks. The canonical use case for reader-writer locking involves very long read-side critical sections, preferably measured in hundreds of microseconds or even milliseconds.

As with exclusive locks, a reader-writer lock acquisition cannot complete until all prior conflicting holders of that lock have released it. If a lock is read-held, then read acquisitions can complete immediately, but write acquisitions must wait until there are no longer any readers holding the lock. If a lock is write-held, then all acquisitions must wait until the writer releases the lock. As with exclusive locks, different reader-writer lock implementations provide different degrees of FIFO ordering to readers on the one hand and to writers on the other.

But suppose a large number of readers hold the lock and a writer is waiting to acquire the lock. Should readers be allowed to continue to acquire the lock, possibly starving the writer? Similarly, suppose that a writer holds the lock and that a large number of both readers and writers are waiting to acquire the lock. When the current writer release the lock, should it be given to a reader or to another writer? If it is given to a reader, how many readers should be allowed to acquire the lock before the next writer is permitted to do so?

There are many possible answers to these questions, with different levels of complexity, overhead, and fairness. Different implementations might have different costs, for example, some types of reader-writer locks incur extremely large latencies when switching from read-holder to write-holder mode. Here are a few possible approaches:

1. Reader-preference implementations unconditionally favor readers over writers, possibly allowing write acquisitions to be indefinitely blocked.

2. Batch-fair implementations ensure that when both readers and writers are acquiring the lock, both have reasonable access via batching. For example, the lock might admit five readers per CPU, then two writers, then five more readers per CPU, and so on.

3. Writer-preference implementations unconditionally favor writers over readers, possibly allowing read acquisitions to be indefinitely blocked.

Of course, these distinctions matter only under conditions of high lock contention.

Please keep the waiting/blocking dual nature of locks firmly in mind. This will be revisited in Chapter 9's discussion of scalable high-performance special-purpose alternatives to locking.

7.2.3 Beyond Reader-Writer Locks

Reader-writer locks and exclusive locks differ in their admission policy: exclusive locks allow at most one holder, while reader-writer locks permit an arbitrary number of read-holders (but only one write-holder). There is a very large number of possible admission policies, one of which is that of the VAX/VMS distributed lock manager (DLM) [ST87], which is shown in Table 7.1. Blank cells indicate compatible modes, while cells containing "X" indicate incompatible modes.

The VAX/VMS DLM uses six modes. For purposes of comparison, exclusive locks use two modes (not held and held), while reader-writer locks use three modes (not held, read held, and write held).

The first mode is null, or not held. This mode is compatible with all other modes, which is to be expected:

Table 7.1: VAX/VMS Distributed Lock Manager Policy

	Null (Not Held)	Concurrent Read	Concurrent Write	Protected Read	Protected Write	Exclusive
Null (Not Held)						
Concurrent Read						X
Concurrent Write				X	X	X
Protected Read			X		X	X
Protected Write			X	X	X	X
Exclusive		X	X	X	X	X

If a thread is not holding a lock, it should not prevent any other thread from acquiring that lock.

The second mode is concurrent read, which is compatible with every other mode except for exclusive. The concurrent-read mode might be used to accumulate approximate statistics on a data structure, while permitting updates to proceed concurrently.

The third mode is concurrent write, which is compatible with null, concurrent read, and concurrent write. The concurrent-write mode might be used to update approximate statistics, while still permitting reads and concurrent updates to proceed concurrently.

The fourth mode is protected read, which is compatible with null, concurrent read, and protected read. The protected-read mode might be used to obtain a consistent snapshot of the data structure, while permitting reads but not updates to proceed concurrently.

The fifth mode is protected write, which is compatible with null and concurrent read. The protected-write mode might be used to carry out updates to a data structure that could interfere with protected readers but which could be tolerated by concurrent readers.

The sixth and final mode is exclusive, which is compatible only with null. The exclusive mode is used when it is necessary to exclude all other accesses.

It is interesting to note that exclusive locks and reader-writer locks can be emulated by the VAX/VMS DLM. Exclusive locks would use only the null and exclusive modes, while reader-writer locks might use the null, protected-read, and protected-write modes.

Quick Quiz 7.19: Is there any other way for the VAX/VMS DLM to emulate a reader-writer lock? ■

Although the VAX/VMS DLM policy has seen widespread production use for distributed databases, it does not appear to be used much in shared-memory applications. One possible reason for this is that the greater communication overheads of distributed databases can hide the greater overhead of the VAX/VMS DLM's more-complex admission policy.

Nevertheless, the VAX/VMS DLM is an interesting illustration of just how flexible the concepts behind locking can be. It also serves as a very simple introduction to the locking schemes used by modern DBMSes, which can have more than thirty locking modes, compared to VAX/VMS's six.

7.2.4 Scoped Locking

The locking primitives discussed thus far require explicit acquisition and release primitives, for example, `spin_lock()` and `spin_unlock()`, respectively. Another approach is to use the object-oriented "resource allocation is initialization" (RAII) pattern [ES90].[4] This pattern is often applied to auto variables in languages like C++, where the corresponding *constructor* is invoked upon entry to the object's scope, and the corresponding *destructor* is invoked upon exit from that scope. This can be applied to locking by having the constructor acquire the lock and the destructor free it.

This approach can be quite useful, in fact in 1990 I was convinced that it was the only type of locking that was needed.[5] One very nice property of RAII locking is that you don't need to carefully release the lock on each and every code path that exits that scope, a property that can eliminate a troublesome set of bugs.

However, RAII locking also has a dark side. RAII makes it quite difficult to encapsulate lock acquisition and release, for example, in iterators. In many iterator implementations, you would like to acquire the lock in the iterator's "start" function and release it in the iterator's "stop" function. RAII locking instead requires that the lock acquisition and release take place in the same level of scoping, making such encapsulation difficult or even impossible.

Strict RAII locking also prohibits overlapping critical sections, due to the fact that scopes must nest. This prohibition makes it difficult or impossible to express a number of useful constructs, for example, locking trees

[4] Though more clearly expressed at `https://www.stroustrup.com/bs_faq2.html#finally`.

[5] My later work with parallelism at Sequent Computer Systems very quickly disabused me of this misguided notion.

www.dbooks.org

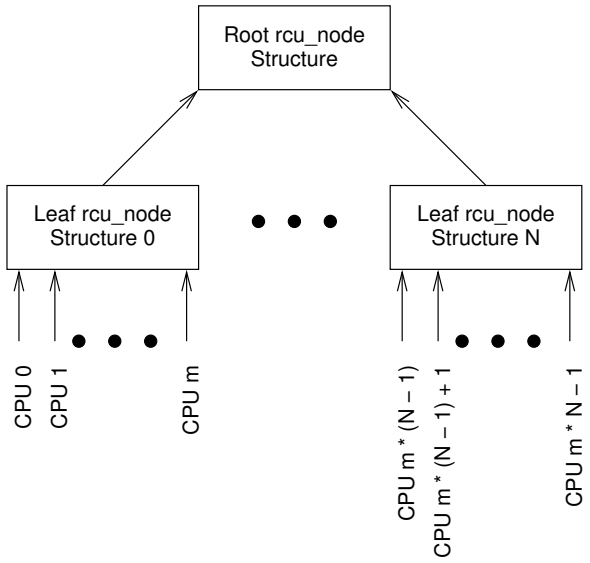


Figure 7.9: Locking Hierarchy

Listing 7.7: Conditional Locking to Reduce Contention

```
1  void force_quiescent_state(struct rcu_node *rnp_leaf)
2  {
3    int ret;
4    struct rcu_node *rnp = rnp_leaf;
5    struct rcu_node *rnp_old = NULL;
6
7    for (; rnp != NULL; rnp = rnp->parent) {
8      ret = (READ_ONCE(gp_flags)) ||
9            !raw_spin_trylock(&rnp->fqslock);
10     if (rnp_old != NULL)
11       raw_spin_unlock(&rnp_old->fqslock);
12     if (ret)
13       return;
14     rnp_old = rnp;
15   }
16   if (!READ_ONCE(gp_flags)) {
17     WRITE_ONCE(gp_flags, 1);
18     do_force_quiescent_state();
19     WRITE_ONCE(gp_flags, 0);
20   }
21   raw_spin_unlock(&rnp_old->fqslock);
22 }
```

that mediate between multiple concurrent attempts to assert an event. Of an arbitrarily large group of concurrent attempts, only one need succeed, and the best strategy for the remaining attempts is for them to fail as quickly and painlessly as possible. Otherwise, lock contention becomes pathological on large systems (where "large" is many hundreds of CPUs). Therefore, C++17 [Smi19] has escapes from strict RAII in its `unique_lock` class, which allows the scope of the critical section to be controlled to roughly the same extent as can be achieved with explicit lock acquisition and release primitives.

Example strict-RAII-unfriendly data structures from Linux-kernel RCU are shown in Figure 7.9. Here, each CPU is assigned a leaf `rcu_node` structure, and each `rcu_node` structure has a pointer to its parent (named, oddly enough, `->parent`), up to the root `rcu_node` structure, which has a `NULL` `->parent` pointer. The number of child `rcu_node` structures per parent can vary, but is typically 32 or 64. Each `rcu_node` structure also contains a lock named `->fqslock`.

The general approach is a *tournament*, where a given CPU conditionally acquires its leaf `rcu_node` structure's `->fqslock`, and, if successful, attempt to acquire that of the parent, then release that of the child. In addition, at each level, the CPU checks a global `gp_flags` variable, and if this variable indicates that some other CPU has asserted the event, the first CPU drops out of the competition. This acquire-then-release sequence continues until either the `gp_flags` variable indicates that someone else won the tournament, one of the attempts to acquire an `->fqslock` fails, or the root `rcu_node` structure's `->fqslock` has been acquired. If the root `rcu_node` structure's `->fqslock` is acquired, a function named `do_force_quiescent_state()` is invoked.

Simplified code to implement this is shown in Listing 7.7. The purpose of this function is to mediate between CPUs who have concurrently detected a need to invoke the `do_force_quiescent_state()` function. At any given time, it only makes sense for one instance of `do_force_quiescent_state()` to be active, so if there are multiple concurrent callers, we need at most one of them to actually invoke `do_force_quiescent_state()`, and we need the rest to (as quickly and painlessly as possible) give up and leave.

To this end, each pass through the loop spanning lines 7–15 attempts to advance up one level in the `rcu_node` hierarchy. If the `gp_flags` variable is already set (line 8) or if the attempt to acquire the current `rcu_node` structure's `->fqslock` is unsuccessful (line 9), then local variable `ret` is set to 1. If line 10 sees that local variable `rnp_old` is non-`NULL`, meaning that we hold `rnp_old`'s `->fqs_lock`, line 11 releases this lock (but only after the attempt has been made to acquire the parent `rcu_node` structure's `->fqslock`). If line 12 sees that either line 8 or 9 saw a reason to give up, line 13 returns to the caller. Otherwise, we must have acquired the current `rcu_node` structure's `->fqslock`, so line 14 saves a pointer to this structure in local variable `rnp_old` in preparation for the next pass through the loop.

If control reaches line 16, we won the tournament, and now holds the root `rcu_node` structure's `->fqslock`. If

line 16 still sees that the global variable gp_flags is zero, line 17 sets gp_flags to one, line 18 invokes do_force_quiescent_state(), and line 19 resets gp_flags back to zero. Either way, line 21 releases the root rcu_node structure's ->fqslock.

Quick Quiz 7.20: The code in Listing 7.7 is ridiculously complicated! Why not conditionally acquire a single global lock? ■

Quick Quiz 7.21: Wait a minute! If we "win" the tournament on line 16 of Listing 7.7, we get to do all the work of do_force_quiescent_state(). Exactly how is that a win, really? ■

This function illustrates the not-uncommon pattern of hierarchical locking. This pattern is difficult to implement using strict RAII locking,[6] just like the iterator encapsulation noted earlier, and so explicit lock/unlock primitives (or C++17-style unique_lock escapes) will be required for the foreseeable future.

7.3 Locking Implementation Issues

> When you translate a dream into reality, it's never a full implementation. It is easier to dream than to do.
>
> *Shai Agassi*

Developers are almost always best-served by using whatever locking primitives are provided by the system, for example, the POSIX pthread mutex locks [Ope97, But97]. Nevertheless, studying sample implementations can be helpful, as can considering the challenges posed by extreme workloads and environments.

7.3.1 Sample Exclusive-Locking Implementation Based on Atomic Exchange

This section reviews the implementation shown in listing 7.8. The data structure for this lock is just an int, as shown on line 1, but could be any integral type. The initial value of this lock is zero, meaning "unlocked", as shown on line 2.

[6] Which is why many RAII locking implementations provide a way to leak the lock out of the scope that it was acquired and into the scope in which it is to be released. However, some object must mediate the scope leaking, which can add complexity compared to non-RAII explicit locking primitives.

Listing 7.8: Sample Lock Based on Atomic Exchange

```
1 typedef int xchglock_t;
2 #define DEFINE_XCHG_LOCK(n) xchglock_t n = 0
3
4 void xchg_lock(xchglock_t *xp)
5 {
6   while (xchg(xp, 1) == 1) {
7     while (READ_ONCE(*xp) == 1)
8       continue;
9   }
10 }
11
12 void xchg_unlock(xchglock_t *xp)
13 {
14   (void)xchg(xp, 0);
15 }
```

Quick Quiz 7.22: Why not rely on the C language's default initialization of zero instead of using the explicit initializer shown on line 2 of Listing 7.8? ■

Lock acquisition is carried out by the xchg_lock() function shown on lines 4–10. This function uses a nested loop, with the outer loop repeatedly atomically exchanging the value of the lock with the value one (meaning "locked"). If the old value was already the value one (in other words, someone else already holds the lock), then the inner loop (lines 7–8) spins until the lock is available, at which point the outer loop makes another attempt to acquire the lock.

Quick Quiz 7.23: Why bother with the inner loop on lines 7–8 of Listing 7.8? Why not simply repeatedly do the atomic exchange operation on line 6? ■

Lock release is carried out by the xchg_unlock() function shown on lines 12–15. Line 14 atomically exchanges the value zero ("unlocked") into the lock, thus marking it as having been released.

Quick Quiz 7.24: Why not simply store zero into the lock word on line 14 of Listing 7.8? ■

This lock is a simple example of a test-and-set lock [SR84], but very similar mechanisms have been used extensively as pure spinlocks in production.

7.3.2 Other Exclusive-Locking Implementations

There are a great many other possible implementations of locking based on atomic instructions, many of which are reviewed in the classic paper by Mellor-Crummey and Scott [MCS91]. These implementations represent different points in a multi-dimensional design tradeoff [GGL+19, Gui18, McK96b]. For example, the atomic-exchange-based test-and-set lock presented in the previous

www.dbooks.org

section works well when contention is low and has the advantage of small memory footprint. It avoids giving the lock to threads that cannot use it, but as a result can suffer from unfairness or even starvation at high contention levels.

In contrast, ticket lock [MCS91], which was once used in the Linux kernel, avoids unfairness at high contention levels. However, as a consequence of its strict FIFO discipline, it can grant the lock to a thread that is currently unable to use it, perhaps due to that thread being preempted or interrupted. On the other hand, it is important to avoid getting too worried about the possibility of preemption and interruption. After all, in many cases, this preemption and interruption could just as well happen just after the lock was acquired.[7]

All locking implementations where waiters spin on a single memory location, including both test-and-set locks and ticket locks, suffer from performance problems at high contention levels. The problem is that the thread releasing the lock must update the value of the corresponding memory location. At low contention, this is not a problem: The corresponding cache line is very likely still local to and writeable by the thread holding the lock. In contrast, at high levels of contention, each thread attempting to acquire the lock will have a read-only copy of the cache line, and the lock holder will need to invalidate all such copies before it can carry out the update that releases the lock. In general, the more CPUs and threads there are, the greater the overhead incurred when releasing the lock under conditions of high contention.

This negative scalability has motivated a number of different queued-lock implementations [And90, GT90, MCS91, WKS94, Cra93, MLH94, TS93], some of which are used in recent versions of the Linux kernel [Cor14b]. Queued locks avoid high cache-invalidation overhead by assigning each thread a queue element. These queue elements are linked together into a queue that governs the order that the lock will be granted to the waiting threads. The key point is that each thread spins on its own queue element, so that the lock holder need only invalidate the first element from the next thread's CPU's cache. This arrangement greatly reduces the overhead of lock handoff at high levels of contention.

More recent queued-lock implementations also take the system's architecture into account, preferentially granting locks locally, while also taking steps to avoid starvation [SSVM02, RH03, RH02, JMRR02, MCM02]. Many of these can be thought of as analogous to the elevator algorithms traditionally used in scheduling disk I/O.

Unfortunately, the same scheduling logic that improves the efficiency of queued locks at high contention also increases their overhead at low contention. Beng-Hong Lim and Anant Agarwal therefore combined a simple test-and-set lock with a queued lock, using the test-and-set lock at low levels of contention and switching to the queued lock at high levels of contention [LA94], thus getting low overhead at low levels of contention and getting fairness and high throughput at high levels of contention. Browning et al. took a similar approach, but avoided the use of a separate flag, so that the test-and-set fast path uses the same sequence of instructions that would be used in a simple test-and-set lock [BMMM05]. This approach has been used in production.

Another issue that arises at high levels of contention is when the lock holder is delayed, especially when the delay is due to preemption, which can result in *priority inversion*, where a low-priority thread holds a lock, but is preempted by a medium priority CPU-bound thread, which results in a high-priority process blocking while attempting to acquire the lock. The result is that the CPU-bound medium-priority process is preventing the high-priority process from running. One solution is *priority inheritance* [LR80], which has been widely used for real-time computing [SRL90, Cor06b], despite some lingering controversy over this practice [Yod04a, Loc02].

Another way to avoid priority inversion is to prevent preemption while a lock is held. Because preventing preemption while locks are held also improves throughput, most proprietary UNIX kernels offer some form of scheduler-conscious synchronization mechanism [KWS97], largely due to the efforts of a certain sizable database vendor. These mechanisms usually take the form of a hint that preemption should be avoided in a given region of code, with this hint typically being placed in a machine register. These hints frequently take the form of a bit set in a particular machine register, which enables extremely low per-lock-acquisition overhead for these mechanisms. In contrast, Linux avoids these hints, instead getting similar results from a mechanism called *futexes* [FRK02, Mol06, Ros06, Dre11].

Interestingly enough, atomic instructions are not strictly needed to implement locks [Dij65, Lam74]. An excellent exposition of the issues surrounding locking implementations based on simple loads and stores may be found in

[7] Besides, the best way of handling high lock contention is to avoid it in the first place! There are nevertheless some situations where high lock contention is the lesser of the available evils, and in any case, studying schemes that deal with high levels of contention is a good mental exercise.

Herlihy's and Shavit's textbook [HS08]. The main point echoed here is that such implementations currently have little practical application, although a careful study of them can be both entertaining and enlightening. Nevertheless, with one exception described below, such study is left as an exercise for the reader.

Gamsa et al. [GKAS99, Section 5.3] describe a token-based mechanism in which a token circulates among the CPUs. When the token reaches a given CPU, it has exclusive access to anything protected by that token. There are any number of schemes that may be used to implement the token-based mechanism, for example:

1. Maintain a per-CPU flag, which is initially zero for all but one CPU. When a CPU's flag is non-zero, it holds the token. When it finishes with the token, it zeroes its flag and sets the flag of the next CPU to one (or to any other non-zero value).
2. Maintain a per-CPU counter, which is initially set to the corresponding CPU's number, which we assume to range from zero to $N - 1$, where N is the number of CPUs in the system. When a CPU's counter is greater than that of the next CPU (taking counter wrap into account), the first CPU holds the token. When it is finished with the token, it sets the next CPU's counter to a value one greater than its own counter.

Quick Quiz 7.25: How can you tell if one counter is greater than another, while accounting for counter wrap? ■

Quick Quiz 7.26: Which is better, the counter approach or the flag approach? ■

This lock is unusual in that a given CPU cannot necessarily acquire it immediately, even if no other CPU is using it at the moment. Instead, the CPU must wait until the token comes around to it. This is useful in cases where CPUs need periodic access to the critical section, but can tolerate variances in token-circulation rate. Gamsa et al. [GKAS99] used it to implement a variant of read-copy update (see Section 9.5), but it could also be used to protect periodic per-CPU operations such as flushing per-CPU caches used by memory allocators [MS93], garbage-collecting per-CPU data structures, or flushing per-CPU data to shared storage (or to mass storage, for that matter).

The Linux kernel now uses queued spinlocks [Cor14b], but because of the complexity of implementations that provide good performance across the range of contention levels, the path has not always been smooth [Mar18, Dea18].

Listing 7.9: Per-Element Locking Without Existence Guarantees

```
1  int delete(int key)
2  {
3    int b;
4    struct element *p;
5
6    b = hashfunction(key);
7    p = hashtable[b];
8    if (p == NULL || p->key != key)
9      return 0;
10   spin_lock(&p->lock);
11   hashtable[b] = NULL;
12   spin_unlock(&p->lock);
13   kfree(p);
14   return 1;
15 }
```

As increasing numbers of people gain familiarity with parallel hardware and parallelize increasing amounts of code, we can continue to expect more special-purpose locking primitives to appear, see for example Guerraoui et al. [GGL+19, Gui18]. Nevertheless, you should carefully consider this important safety tip: Use the standard synchronization primitives whenever humanly possible. The big advantage of the standard synchronization primitives over roll-your-own efforts is that the standard primitives are typically *much* less bug-prone.[8]

7.4 Lock-Based Existence Guarantees

> Existence precedes and rules essence.
>
> *Jean-Paul Sartre*

A key challenge in parallel programming is to provide *existence guarantees* [GKAS99], so that attempts to access a given object can rely on that object being in existence throughout a given access attempt. In some cases, existence guarantees are implicit:

1. Global variables and static local variables in the base module will exist as long as the application is running.
2. Global variables and static local variables in a loaded module will exist as long as that module remains loaded.

[8] And yes, I have done at least my share of roll-your-own synchronization primitives. However, you will notice that my hair is much greyer than it was before I started doing that sort of work. Coincidence? Maybe. But are you *really* willing to risk your own hair turning prematurely grey?

www.dbooks.org

3. A module will remain loaded as long as at least one of its functions has an active instance.

4. A given function instance's on-stack variables will exist until that instance returns.

5. If you are executing within a given function or have been called (directly or indirectly) from that function, then the given function has an active instance.

These implicit existence guarantees are straightforward, though bugs involving implicit existence guarantees really can happen.

Quick Quiz 7.27: How can relying on implicit existence guarantees result in a bug? ■

But the more interesting—and troublesome—guarantee involves heap memory: A dynamically allocated data structure will exist until it is freed. The problem to be solved is to synchronize the freeing of the structure with concurrent accesses to that same structure. One way to do this is with *explicit guarantees*, such as locking. If a given structure may only be freed while holding a given lock, then holding that lock guarantees that structure's existence.

But this guarantee depends on the existence of the lock itself. One straightforward way to guarantee the lock's existence is to place the lock in a global variable, but global locking has the disadvantage of limiting scalability. One way of providing scalability that improves as the size of the data structure increases is to place a lock in each element of the structure. Unfortunately, putting the lock that is to protect a data element in the data element itself is subject to subtle race conditions, as shown in Listing 7.9.

Quick Quiz 7.28: What if the element we need to delete is not the first element of the list on line 8 of Listing 7.9? ■

To see one of these race conditions, consider the following sequence of events:

1. Thread 0 invokes `delete(0)`, and reaches line 10 of the listing, acquiring the lock.

2. Thread 1 concurrently invokes `delete(0)`, reaching line 10, but spins on the lock because Thread 0 holds it.

3. Thread 0 executes lines 11–14, removing the element from the hashtable, releasing the lock, and then freeing the element.

Listing 7.10: Per-Element Locking With Lock-Based Existence Guarantees

```
1  int delete(int key)
2  {
3    int b;
4    struct element *p;
5    spinlock_t *sp;
6
7    b = hashfunction(key);
8    sp = &locktable[b];
9    spin_lock(sp);
10   p = hashtable[b];
11   if (p == NULL || p->key != key) {
12     spin_unlock(sp);
13     return 0;
14   }
15   hashtable[b] = NULL;
16   spin_unlock(sp);
17   kfree(p);
18   return 1;
19 }
```

4. Thread 0 continues execution, and allocates memory, getting the exact block of memory that it just freed.

5. Thread 0 then initializes this block of memory as some other type of structure.

6. Thread 1's `spin_lock()` operation fails due to the fact that what it believes to be `p->lock` is no longer a spinlock.

Because there is no existence guarantee, the identity of the data element can change while a thread is attempting to acquire that element's lock on line 10!

One way to fix this example is to use a hashed set of global locks, so that each hash bucket has its own lock, as shown in Listing 7.10. This approach allows acquiring the proper lock (on line 9) before gaining a pointer to the data element (on line 10). Although this approach works quite well for elements contained in a single partitionable data structure such as the hash table shown in the listing, it can be problematic if a given data element can be a member of multiple hash tables or given more-complex data structures such as trees or graphs. Not only can these problems be solved, but the solutions also form the basis of lock-based software transactional memory implementations [ST95, DSS06]. However, Chapter 9 describes simpler—and faster—ways of providing existence guarantees.

7.5 Locking: Hero or Villain?

> You either die a hero or live long enough to become the villain.
>
> *Aaron Eckhart*

As is often the case in real life, locking can be either hero or villain, depending on how it is used and on the problem at hand. In my experience, those writing whole applications are happy with locking, those writing parallel libraries are less happy, and those parallelizing existing sequential libraries are extremely unhappy. The following sections discuss some reasons for these differences in viewpoints.

7.5.1 Locking For Applications: Hero!

When writing an entire application (or entire kernel), developers have full control of the design, including the synchronization design. Assuming that the design makes good use of partitioning, as discussed in Chapter 6, locking can be an extremely effective synchronization mechanism, as demonstrated by the heavy use of locking in production-quality parallel software.

Nevertheless, although such software usually bases most of its synchronization design on locking, such software also almost always makes use of other synchronization mechanisms, including special counting algorithms (Chapter 5), data ownership (Chapter 8), reference counting (Section 9.2), hazard pointers (Section 9.3), sequence locking (Section 9.4), and read-copy update (Section 9.5). In addition, practitioners use tools for deadlock detection [Cor06a], lock acquisition/release balancing [Cor04b], cache-miss analysis [The11], hardware-counter-based profiling [EGMdB11, The12b], and many more besides.

Given careful design, use of a good combination of synchronization mechanisms, and good tooling, locking works quite well for applications and kernels.

7.5.2 Locking For Parallel Libraries: Just Another Tool

Unlike applications and kernels, the designer of a library cannot know the locking design of the code that the library will be interacting with. In fact, that code might not be written for years to come. Library designers therefore have less control and must exercise more care when laying out their synchronization design.

Deadlock is of course of particular concern, and the techniques discussed in Section 7.1.1 need to be applied. One popular deadlock-avoidance strategy is therefore to ensure that the library's locks are independent subtrees of the enclosing program's locking hierarchy. However, this can be harder than it looks.

One complication was discussed in Section 7.1.1.2, namely when library functions call into application code, with `qsort()`'s comparison-function argument being a case in point. Another complication is the interaction with signal handlers. If an application signal handler is invoked from a signal received within the library function, deadlock can ensue just as surely as if the library function had called the signal handler directly. A final complication occurs for those library functions that can be used between a `fork()`/`exec()` pair, for example, due to use of the `system()` function. In this case, if your library function was holding a lock at the time of the `fork()`, then the child process will begin life with that lock held. Because the thread that will release the lock is running in the parent but not the child, if the child calls your library function, deadlock will ensue.

The following strategies may be used to avoid deadlock problems in these cases:

1. Don't use either callbacks or signals.
2. Don't acquire locks from within callbacks or signal handlers.
3. Let the caller control synchronization.
4. Parameterize the library API to delegate locking to caller.
5. Explicitly avoid callback deadlocks.
6. Explicitly avoid signal-handler deadlocks.
7. Avoid invoking `fork()`

Each of these strategies is discussed in one of the following sections.

7.5.2.1 Use Neither Callbacks Nor Signals

If a library function avoids callbacks and the application as a whole avoids signals, then any locks acquired by that library function will be leaves of the locking-hierarchy tree. This arrangement avoids deadlock, as discussed in Section 7.1.1.1. Although this strategy works extremely well where it applies, there are some applications that

www.dbooks.org

must use signal handlers, and there are some library functions (such as the `qsort()` function discussed in Section 7.1.1.2) that require callbacks.

The strategy described in the next section can often be used in these cases.

7.5.2.2 Avoid Locking in Callbacks and Signal Handlers

If neither callbacks nor signal handlers acquire locks, then they cannot be involved in deadlock cycles, which allows straightforward locking hierarchies to once again consider library functions to be leaves on the locking-hierarchy tree. This strategy works very well for most uses of `qsort`, whose callbacks usually simply compare the two values passed in to them. This strategy also works wonderfully for many signal handlers, especially given that acquiring locks from within signal handlers is generally frowned upon [Gro01],[9] but can fail if the application needs to manipulate complex data structures from a signal handler.

Here are some ways to avoid acquiring locks in signal handlers even if complex data structures must be manipulated:

1. Use simple data structures based on non-blocking synchronization, as will be discussed in Section 14.2.1.

2. If the data structures are too complex for reasonable use of non-blocking synchronization, create a queue that allows non-blocking enqueue operations. In the signal handler, instead of manipulating the complex data structure, add an element to the queue describing the required change. A separate thread can then remove elements from the queue and carry out the required changes using normal locking. There are a number of readily available implementations of concurrent queues [KLP12, Des09b, MS96].

This strategy should be enforced with occasional manual or (preferably) automated inspections of callbacks and signal handlers. When carrying out these inspections, be wary of clever coders who might have (unwisely) created home-brew locks from atomic operations.

7.5.2.3 Caller Controls Synchronization

Letting the caller control synchronization works extremely well when the library functions are operating on independent caller-visible instances of a data structure, each of which may be synchronized separately. For example, if the library functions operate on a search tree, and if the application needs a large number of independent search trees, then the application can associate a lock with each tree. The application then acquires and releases locks as needed, so that the library need not be aware of parallelism at all. Instead, the application controls the parallelism, so that locking can work very well, as was discussed in Section 7.5.1.

However, this strategy fails if the library implements a data structure that requires internal concurrency, for example, a hash table or a parallel sort. In this case, the library absolutely must control its own synchronization.

7.5.2.4 Parameterize Library Synchronization

The idea here is to add arguments to the library's API to specify which locks to acquire, how to acquire and release them, or both. This strategy allows the application to take on the global task of avoiding deadlock by specifying which locks to acquire (by passing in pointers to the locks in question) and how to acquire them (by passing in pointers to lock acquisition and release functions), but also allows a given library function to control its own concurrency by deciding where the locks should be acquired and released.

In particular, this strategy allows the lock acquisition and release functions to block signals as needed without the library code needing to be concerned with which signals need to be blocked by which locks. The separation of concerns used by this strategy can be quite effective, but in some cases the strategies laid out in the following sections can work better.

That said, passing explicit pointers to locks to external APIs must be very carefully considered, as discussed in Section 7.1.1.4. Although this practice is sometimes the right thing to do, you should do yourself a favor by looking into alternative designs first.

7.5.2.5 Explicitly Avoid Callback Deadlocks

The basic rule behind this strategy was discussed in Section 7.1.1.2: "Release all locks before invoking unknown code." This is usually the best approach because it allows the application to ignore the library's locking hierarchy: the library remains a leaf or isolated subtree of the application's overall locking hierarchy.

In cases where it is not possible to release all locks before invoking unknown code, the layered locking hierarchies described in Section 7.1.1.3 can work well. For example, if

[9] But the standard's words do not stop clever coders from creating their own home-brew locking primitives from atomic operations.

the unknown code is a signal handler, this implies that the library function block signals across all lock acquisitions, which can be complex and slow. Therefore, in cases where signal handlers (probably unwisely) acquire locks, the strategies in the next section may prove helpful.

7.5.2.6 Explicitly Avoid Signal-Handler Deadlocks

Suppose that a given library function is known to acquire locks, but does not block signals. Suppose further that it is necessary to invoke that function both from within and outside of a signal handler, and that it is not permissible to modify this library function. Of course, if no special action is taken, then if a signal arrives while that library function is holding its lock, deadlock can occur when the signal handler invokes that same library function, which in turn attempts to re-acquire that same lock.

Such deadlocks can be avoided as follows:

1. If the application invokes the library function from within a signal handler, then that signal must be blocked every time that the library function is invoked from outside of a signal handler.

2. If the application invokes the library function while holding a lock acquired within a given signal handler, then that signal must be blocked every time that the library function is called outside of a signal handler.

These rules can be enforced by using tools similar to the Linux kernel's lockdep lock dependency checker [Cor06a]. One of the great strengths of lockdep is that it is not fooled by human intuition [Ros11].

7.5.2.7 Library Functions Used Between `fork()` and `exec()`

As noted earlier, if a thread executing a library function is holding a lock at the time that some other thread invokes `fork()`, the fact that the parent's memory is copied to create the child means that this lock will be born held in the child's context. The thread that will release this lock is running in the parent, but not in the child, which means that the child's copy of this lock will never be released. Therefore, any attempt on the part of the child to invoke that same library function will result in deadlock.

A pragmatic and straightforward way of solving this problem is to `fork()` a child process while the process is still single-threaded, and have this child process remain single-threaded. Requests to create further child processes can then be communicated to this initial child process, which can safely carry out any needed `fork()` and `exec()` system calls on behalf of its multi-threaded parent process.

Another rather less pragmatic and straightforward solution to this problem is to have the library function check to see if the owner of the lock is still running, and if not, "breaking" the lock by re-initializing and then acquiring it. However, this approach has a couple of vulnerabilities:

1. The data structures protected by that lock are likely to be in some intermediate state, so that naively breaking the lock might result in arbitrary memory corruption.

2. If the child creates additional threads, two threads might break the lock concurrently, with the result that both threads believe they own the lock. This could again result in arbitrary memory corruption.

The `pthread_atfork()` function is provided to help deal with these situations. The idea is to register a triplet of functions, one to be called by the parent before the `fork()`, one to be called by the parent after the `fork()`, and one to be called by the child after the `fork()`. Appropriate cleanups can then be carried out at these three points.

Be warned, however, that coding of `pthread_atfork()` handlers is quite subtle in general. The cases where `pthread_atfork()` works best are cases where the data structure in question can simply be re-initialized by the child.

7.5.2.8 Parallel Libraries: Discussion

Regardless of the strategy used, the description of the library's API must include a clear description of that strategy and how the caller should interact with that strategy. In short, constructing parallel libraries using locking is possible, but not as easy as constructing a parallel application.

7.5.3 Locking For Parallelizing Sequential Libraries: Villain!

With the advent of readily available low-cost multicore systems, a common task is parallelizing an existing library that was designed with only single-threaded use in mind. This all-too-common disregard for parallelism can result in a library API that is severely flawed from a parallel-programming viewpoint. Candidate flaws include:

1. Implicit prohibition of partitioning.

www.dbooks.org

2. Callback functions requiring locking.

3. Object-oriented spaghetti code.

These flaws and the consequences for locking are discussed in the following sections.

7.5.3.1 Partitioning Prohibited

Suppose that you were writing a single-threaded hash-table implementation. It is easy and fast to maintain an exact count of the total number of items in the hash table, and also easy and fast to return this exact count on each addition and deletion operation. So why not?

One reason is that exact counters do not perform or scale well on multicore systems, as was seen in Chapter 5. As a result, the parallelized implementation of the hash table will not perform or scale well.

So what can be done about this? One approach is to return an approximate count, using one of the algorithms from Chapter 5. Another approach is to drop the element count altogether.

Either way, it will be necessary to inspect uses of the hash table to see why the addition and deletion operations need the exact count. Here are a few possibilities:

1. Determining when to resize the hash table. In this case, an approximate count should work quite well. It might also be useful to trigger the resizing operation from the length of the longest chain, which can be computed and maintained in a nicely partitioned per-chain manner.

2. Producing an estimate of the time required to traverse the entire hash table. An approximate count works well in this case, also.

3. For diagnostic purposes, for example, to check for items being lost when transferring them to and from the hash table. This clearly requires an exact count. However, given that this usage is diagnostic in nature, it might suffice to maintain the lengths of the hash chains, then to infrequently sum them up while locking out addition and deletion operations.

It turns out that there is now a strong theoretical basis for some of the constraints that performance and scalability place on a parallel library's APIs [AGH+11a, AGH+11b, McK11b]. Anyone designing a parallel library needs to pay close attention to those constraints.

Although it is all too easy to blame locking for what are really problems due to a concurrency-unfriendly API, doing so is not helpful. On the other hand, one has little choice but to sympathize with the hapless developer who made this choice in (say) 1985. It would have been a rare and courageous developer to anticipate the need for parallelism at that time, and it would have required an even more rare combination of brilliance and luck to actually arrive at a good parallel-friendly API.

Times change, and code must change with them. That said, there might be a huge number of users of a popular library, in which case an incompatible change to the API would be quite foolish. Adding a parallel-friendly API to complement the existing heavily used sequential-only API is usually the best course of action.

Nevertheless, human nature being what it is, we can expect our hapless developer to be more likely to complain about locking than about his or her own poor (though understandable) API design choices.

7.5.3.2 Deadlock-Prone Callbacks

Sections 7.1.1.2, 7.1.1.3, and 7.5.2 described how undisciplined use of callbacks can result in locking woes. These sections also described how to design your library function to avoid these problems, but it is unrealistic to expect a 1990s programmer with no experience in parallel programming to have followed such a design. Therefore, someone attempting to parallelize an existing callback-heavy single-threaded library will likely have many opportunities to curse locking's villainy.

If there are a very large number of uses of a callback-heavy library, it may be wise to again add a parallel-friendly API to the library in order to allow existing users to convert their code incrementally. Alternatively, some advocate use of transactional memory in these cases. While the jury is still out on transactional memory, Section 17.2 discusses its strengths and weaknesses. It is important to note that hardware transactional memory (discussed in Section 17.3) cannot help here unless the hardware transactional memory implementation provides forward-progress guarantees, which few do. Other alternatives that appear to be quite practical (if less heavily hyped) include the methods discussed in Sections 7.1.1.5, and 7.1.1.6, as well as those that will be discussed in Chapters 8 and 9.

7.5.3.3 Object-Oriented Spaghetti Code

Object-oriented programming went mainstream sometime in the 1980s or 1990s, and as a result there is a huge amount of single-threaded object-oriented code in production.

Although object orientation can be a valuable software technique, undisciplined use of objects can easily result in object-oriented spaghetti code. In object-oriented spaghetti code, control flits from object to object in an essentially random manner, making the code hard to understand and even harder, and perhaps impossible, to accommodate a locking hierarchy.

Although many might argue that such code should be cleaned up in any case, such things are much easier to say than to do. If you are tasked with parallelizing such a beast, you can reduce the number of opportunities to curse locking by using the techniques described in Sections 7.1.1.5, and 7.1.1.6, as well as those that will be discussed in Chapters 8 and 9. This situation appears to be the use case that inspired transactional memory, so it might be worth a try as well. That said, the choice of synchronization mechanism should be made in light of the hardware habits discussed in Chapter 3. After all, if the overhead of the synchronization mechanism is orders of magnitude more than that of the operations being protected, the results are not going to be pretty.

And that leads to a question well worth asking in these situations: Should the code remain sequential? For example, perhaps parallelism should be introduced at the process level rather than the thread level. In general, if a task is proving extremely hard, it is worth some time spent thinking about not only alternative ways to accomplish that particular task, but also alternative tasks that might better solve the problem at hand.

7.6 Summary

> Achievement unlocked.
>
> *Unknown*

Locking is perhaps the most widely used and most generally useful synchronization tool. However, it works best when designed into an application or library from the beginning. Given the large quantity of pre-existing single-threaded code that might need to one day run in parallel, locking should therefore not be the only tool in your parallel-programming toolbox. The next few chapters will discuss other tools, and how they can best be used in concert with locking and with each other.

www.dbooks.org

It is mine, I tell you. My own. My precious. Yes, my precious.

Gollum in "The Fellowship of the Ring", J.R.R. Tolkien

Chapter 8

Data Ownership

One of the simplest ways to avoid the synchronization overhead that comes with locking is to parcel the data out among the threads (or, in the case of kernels, CPUs) so that a given piece of data is accessed and modified by only one of the threads. Interestingly enough, data ownership covers each of the "big three" parallel design techniques: It partitions over threads (or CPUs, as the case may be), it batches all local operations, and its elimination of synchronization operations is weakening carried to its logical extreme. It should therefore be no surprise that data ownership is heavily used: Even novices use it almost instinctively. In fact, it is used so heavily that this chapter will not introduce any new examples, but will instead reference examples from previous chapters.

Quick Quiz 8.1: What form of data ownership is extremely difficult to avoid when creating shared-memory parallel programs (for example, using pthreads) in C or C++? ■

There are a number of approaches to data ownership. Section 8.1 presents the logical extreme in data ownership, where each thread has its own private address space. Section 8.2 looks at the opposite extreme, where the data is shared, but different threads own different access rights to the data. Section 8.3 describes function shipping, which is a way of allowing other threads to have indirect access to data owned by a particular thread. Section 8.4 describes how designated threads can be assigned ownership of a specified function and the related data. Section 8.5 discusses improving performance by transforming algorithms with shared data to instead use data ownership. Finally, Section 8.6 lists a few software environments that feature data ownership as a first-class citizen.

8.1 Multiple Processes

A man's home is his castle

Ancient Laws of England

Section 4.1 introduced the following example:

```
1 compute_it 1 > compute_it.1.out &
2 compute_it 2 > compute_it.2.out &
3 wait
4 cat compute_it.1.out
5 cat compute_it.2.out
```

This example runs two instances of the `compute_it` program in parallel, as separate processes that do not share memory. Therefore, all data in a given process is owned by that process, so that almost the entirety of data in the above example is owned. This approach almost entirely eliminates synchronization overhead. The resulting combination of extreme simplicity and optimal performance is obviously quite attractive.

Quick Quiz 8.2: What synchronization remains in the example shown in Section 8.1? ■

Quick Quiz 8.3: Is there any shared data in the example shown in Section 8.1? ■

This same pattern can be written in C as well as in `sh`, as illustrated by Listings 4.1 and 4.2.

It bears repeating that these trivial forms of parallelism are not in any way cheating or ducking responsibility, but are rather simple and elegant ways to make your code run faster. It is fast, scales well, is easy to program, easy to maintain, and gets the job done. In addition, taking this approach (where applicable) allows the developer more time to focus on other things whether these things

Edition.2-rc11

www.dbooks.org

might involve applying sophisticated single-threaded optimizations to `compute_it` on the one hand, or applying sophisticated parallel-programming patterns to portions of the code where this approach is inapplicable. What is not to like?

The next section discusses the use of data ownership in shared-memory parallel programs.

8.2 Partial Data Ownership and pthreads

> Give thy mind more to what thou hast than to what thou hast not.
>
> *Marcus Aurelius Antoninus*

Concurrent counting (see Chapter 5) uses data ownership heavily, but adds a twist. Threads are not allowed to modify data owned by other threads, but they are permitted to read it. In short, the use of shared memory allows more nuanced notions of ownership and access rights.

For example, consider the per-thread statistical counter implementation shown in Listing 5.4 on page 53. Here, `inc_count()` updates only the corresponding thread's instance of `counter`, while `read_count()` accesses, but does not modify, all threads' instances of `counter`.

Quick Quiz 8.4: Does it ever make sense to have partial data ownership where each thread reads only its own instance of a per-thread variable, but writes to other threads' instances? ■

Partial data ownership is also common within the Linux kernel. For example, a given CPU might be permitted to read a given set of its own per-CPU variables only with interrupts disabled, another CPU might be permitted to read that same set of the first CPU's per-CPU variables only when holding the corresponding per-CPU lock. Then that given CPU would be permitted to update this set of its own per-CPU variables if it both has interrupts disabled and holds its per-CPU lock. This arrangement can be thought of as a reader-writer lock that allows each CPU very low-overhead access to its own set of per-CPU variables. There are a great many variations on this theme.

For its own part, pure data ownership is also both common and useful, for example, the per-thread memory-allocator caches discussed in Section 6.4.3 starting on page 88. In this algorithm, each thread's cache is completely private to that thread.

8.3 Function Shipping

> If the mountain will not come to Muhammad, then Muhammad must go to the mountain.
>
> *Essays, Francis Bacon*

The previous section described a weak form of data ownership where threads reached out to other threads' data. This can be thought of as bringing the data to the functions that need it. An alternative approach is to send the functions to the data.

Such an approach is illustrated in Section 5.4.3 beginning on page 64, in particular the `flush_local_count_sig()` and `flush_local_count()` functions in Listing 5.18 on page 66.

The `flush_local_count_sig()` function is a signal handler that acts as the shipped function. The `pthread_kill()` function in `flush_local_count()` sends the signal—shipping the function—and then waits until the shipped function executes. This shipped function has the not-unusual added complication of needing to interact with any concurrently executing `add_count()` or `sub_count()` functions (see Listing 5.19 on page 66 and Listing 5.20 on page 67).

Quick Quiz 8.5: What mechanisms other than POSIX signals may be used for function shipping? ■

8.4 Designated Thread

> Let a man practice the profession which he best knows.
>
> *Cicero*

The earlier sections describe ways of allowing each thread to keep its own copy or its own portion of the data. In contrast, this section describes a functional-decomposition approach, where a special designated thread owns the rights to the data that is required to do its job. The eventually consistent counter implementation described in Section 5.2.4 provides an example. This implementation has a designated thread that runs the `eventual()` function shown on lines 17–34 of Listing 5.5. This `eventual()` thread periodically pulls the per-thread counts into the global counter, so that accesses to the global counter will, as the name says, eventually converge on the actual value.

Quick Quiz 8.6: But none of the data in the `eventual()` function shown on lines 17–34 of Listing 5.5 is actually owned by the `eventual()` thread! In just what way is this data ownership??? ■

8.5 Privatization

> There is, of course, a difference between what a man seizes and what he really possesses.
>
> *Pearl S. Buck*

One way of improving the performance and scalability of a shared-memory parallel program is to transform it so as to convert shared data to private data that is owned by a particular thread.

An excellent example of this is shown in the answer to one of the Quick Quizzes in Section 6.1.1, which uses privatization to produce a solution to the Dining Philosophers problem with much better performance and scalability than that of the standard textbook solution. The original problem has five philosophers sitting around the table with one fork between each adjacent pair of philosophers, which permits at most two philosophers to eat concurrently.

We can trivially privatize this problem by providing an additional five forks, so that each philosopher has his or her own private pair of forks. This allows all five philosophers to eat concurrently, and also offers a considerable reduction in the spread of certain types of disease.

In other cases, privatization imposes costs. For example, consider the simple limit counter shown in Listing 5.7 on page 57. This is an example of an algorithm where threads can read each others' data, but are only permitted to update their own data. A quick review of the algorithm shows that the only cross-thread accesses are in the summation loop in `read_count()`. If this loop is eliminated, we move to the more-efficient pure data ownership, but at the cost of a less-accurate result from `read_count()`.

Quick Quiz 8.7: Is it possible to obtain greater accuracy while still maintaining full privacy of the per-thread data? ■

Partial privatization is also possible, with some synchronization requirements, but less than in the fully shared case. Some partial-privatization possibilities were explored in Section 4.3.4.4. Chapter 9 will introduce a temporal component to data ownership by providing ways of safely taking public data structures private.

In short, privatization is a powerful tool in the parallel programmer's toolbox, but it must nevertheless be used with care. Just like every other synchronization primitive, it has the potential to increase complexity while decreasing performance and scalability.

8.6 Other Uses of Data Ownership

> Everything comes to us that belongs to us if we create the capacity to receive it.
>
> *Rabindranath Tagore*

Data ownership works best when the data can be partitioned so that there is little or no need for cross thread access or update. Fortunately, this situation is reasonably common, and in a wide variety of parallel-programming environments.

Examples of data ownership include:

1. All message-passing environments, such as MPI [MPI08] and BOINC [Uni08a].
2. Map-reduce [Jac08].
3. Client-server systems, including RPC, web services, and pretty much any system with a back-end database server.
4. Shared-nothing database systems.
5. Fork-join systems with separate per-process address spaces.
6. Process-based parallelism, such as the Erlang language.
7. Private variables, for example, C-language on-stack auto variables, in threaded environments.
8. Many parallel linear-algebra algorithms, especially those well-suited for GPGPUs.[1]
9. Operating-system kernels adapted for networking, where each connection (also called *flow* [DKS89, Zha89, McK90]) is assigned to a specific thread. One recent example of this approach is the IX operating system [BPP+16]. IX does have some shared data structures, which use synchronization mechanisms to be described in Section 9.5.

[1] But note that a great many other classes of applications have also been ported to GPGPUs [Mat17, AMD20, NVi17a, NVi17b].

www.dbooks.org

Data ownership is perhaps the most underappreciated synchronization mechanism in existence. When used properly, it delivers unrivaled simplicity, performance, and scalability. Perhaps its simplicity costs it the respect that it deserves. Hopefully a greater appreciation for the subtlety and power of data ownership will lead to greater level of respect, to say nothing of leading to greater performance and scalability coupled with reduced complexity.

All things come to those who wait.

Violet Fane

Chapter 9

Deferred Processing

The strategy of deferring work goes back before the dawn of recorded history. It has occasionally been derided as procrastination or even as sheer laziness. However, in the last few decades workers have recognized this strategy's value in simplifying and streamlining parallel algorithms [KL80, Mas92]. Believe it or not, "laziness" in parallel programming often outperforms and out-scales industriousness! These performance and scalability benefits stem from the fact that deferring work can enable weakening of synchronization primitives, thereby reducing synchronization overhead. General approaches of work deferral include reference counting (Section 9.2), hazard pointers (Section 9.3), sequence locking (Section 9.4), and RCU (Section 9.5). Finally, Section 9.6 describes how to choose among the work-deferral schemes covered in this chapter and Section 9.7 discusses updates. But first, Section 9.1 will introduce an example algorithm that will be used to compare and contrast these approaches.

9.1 Running Example

An ounce of application is worth a ton of abstraction.

Booker T. Washington

This chapter will use a simplified packet-routing algorithm to demonstrate the value of these approaches and to allow them to be compared. Routing algorithms are used in operating-system kernels to deliver each outgoing TCP/IP packets to the appropriate network interface. This particular algorithm is a simplified version of the classic 1980s packet-train-optimized algorithm used in BSD UNIX [Jac88], consisting of a simple linked list.[1] Modern routing algorithms use more complex data structures, however a simple algorithm will help highlight issues specific to parallelism in a straightforward setting.

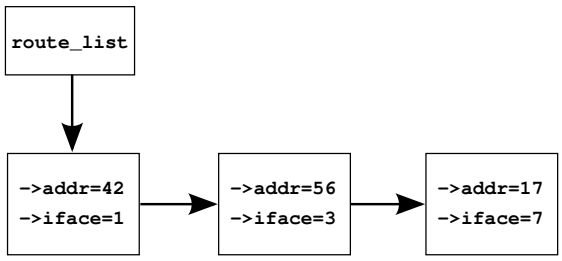


Figure 9.1: Pre-BSD Packet Routing List

We further simplify the algorithm by reducing the search key from a quadruple consisting of source and destination IP addresses and ports all the way down to a simple integer. The value looked up and returned will also be a simple integer, so that the data structure is as shown in Figure 9.1, which directs packets with address 42 to interface 1, address 56 to interface 3, and address 17 to interface 7. This list will normally be searched frequently and updated rarely. In Chapter 3 we learned that the best ways to evade inconvenient laws of physics, such as the finite speed of light and the atomic nature of matter, is to either partition the data or to rely on read-mostly sharing. This chapter applies read-mostly sharing techniques to Pre-BSD packet routing.

Listing 9.1 (`route_seq.c`) shows a simple single-threaded implementation corresponding to Figure 9.1. Lines 1–5 define a `route_entry` structure and line 6 defines the `route_list` header. Lines 8–20 define `route_lookup()`, which sequentially searches `route_list`, returning the corresponding `->iface`, or `ULONG_MAX` if there is no such route entry. Lines 22–33 define `route_`

[1] In other words, this is not OpenBSD, NetBSD, or even FreeBSD, but none other than Pre-BSD.

Edition.2-rc11

www.dbooks.org

Listing 9.1: Sequential Pre-BSD Routing Table

```
1 struct route_entry {
2   struct cds_list_head re_next;
3   unsigned long addr;
4   unsigned long iface;
5 };
6 CDS_LIST_HEAD(route_list);
7
8 unsigned long route_lookup(unsigned long addr)
9 {
10  struct route_entry *rep;
11  unsigned long ret;
12
13  cds_list_for_each_entry(rep, &route_list, re_next) {
14    if (rep->addr == addr) {
15      ret = rep->iface;
16      return ret;
17    }
18  }
19  return ULONG_MAX;
20 }
21
22 int route_add(unsigned long addr, unsigned long interface)
23 {
24  struct route_entry *rep;
25
26  rep = malloc(sizeof(*rep));
27  if (!rep)
28    return -ENOMEM;
29  rep->addr = addr;
30  rep->iface = interface;
31  cds_list_add(&rep->re_next, &route_list);
32  return 0;
33 }
34
35 int route_del(unsigned long addr)
36 {
37  struct route_entry *rep;
38
39  cds_list_for_each_entry(rep, &route_list, re_next) {
40    if (rep->addr == addr) {
41      cds_list_del(&rep->re_next);
42      free(rep);
43      return 0;
44    }
45  }
46  return -ENOENT;
47 }
```

Listing 9.2: Reference-Counted Pre-BSD Routing Table Lookup (BUGGY!!!)

```
1 struct route_entry {
2   atomic_t re_refcnt;
3   struct route_entry *re_next;
4   unsigned long addr;
5   unsigned long iface;
6   int re_freed;
7 };
8 struct route_entry route_list;
9 DEFINE_SPINLOCK(routelock);
10
11 static void re_free(struct route_entry *rep)
12 {
13   WRITE_ONCE(rep->re_freed, 1);
14   free(rep);
15 }
16
17 unsigned long route_lookup(unsigned long addr)
18 {
19   int old;
20   int new;
21   struct route_entry *rep;
22   struct route_entry **repp;
23   unsigned long ret;
24
25 retry:
26   repp = &route_list.re_next;
27   rep = NULL;
28   do {
29     if (rep && atomic_dec_and_test(&rep->re_refcnt))
30       re_free(rep);
31     rep = READ_ONCE(*repp);
32     if (rep == NULL)
33       return ULONG_MAX;
34     do {
35       if (READ_ONCE(rep->re_freed))
36         abort();
37       old = atomic_read(&rep->re_refcnt);
38       if (old <= 0)
39         goto retry;
40       new = old + 1;
41     } while (atomic_cmpxchg(&rep->re_refcnt,
42                             old, new) != old);
43     repp = &rep->re_next;
44   } while (rep->addr != addr);
45   ret = rep->iface;
46   if (atomic_dec_and_test(&rep->re_refcnt))
47     re_free(rep);
48   return ret;
49 }
```

`add()`, which allocates a `route_entry` structure, initializes it, and adds it to the list, returning `-ENOMEM` in case of memory-allocation failure. Finally, lines 35–47 define `route_del()`, which removes and frees the specified `route_entry` structure if it exists, or returns `-ENOENT` otherwise.

This single-threaded implementation serves as a prototype for the various concurrent implementations in this chapter, and also as an estimate of ideal scalability and performance.

9.2 Reference Counting

> I am never letting you go!
>
> Unknown

Reference counting tracks the number of references to a given object in order to prevent that object from being prematurely freed. As such, it has a long and honorable history of use dating back to at least an early 1960s Weizenbaum paper [Wei63]. Weizenbaum discusses reference counting as if it was already well-known, so it likely dates back to the 1950s or even to the 1940s. And perhaps

Listing 9.3: Reference-Counted Pre-BSD Routing Table Add/Delete (BUGGY!!!)

```
1  int route_add(unsigned long addr, unsigned long interface)
2  {
3    struct route_entry *rep;
4
5    rep = malloc(sizeof(*rep));
6    if (!rep)
7      return -ENOMEM;
8    atomic_set(&rep->re_refcnt, 1);
9    rep->addr = addr;
10   rep->iface = interface;
11   spin_lock(&routelock);
12   rep->re_next = route_list.re_next;
13   rep->re_freed = 0;
14   route_list.re_next = rep;
15   spin_unlock(&routelock);
16   return 0;
17 }
18
19 int route_del(unsigned long addr)
20 {
21   struct route_entry *rep;
22   struct route_entry **repp;
23
24   spin_lock(&routelock);
25   repp = &route_list.re_next;
26   for (;;) {
27     rep = *repp;
28     if (rep == NULL)
29       break;
30     if (rep->addr == addr) {
31       *repp = rep->re_next;
32       spin_unlock(&routelock);
33       if (atomic_dec_and_test(&rep->re_refcnt))
34         re_free(rep);
35       return 0;
36     }
37     repp = &rep->re_next;
38   }
39   spin_unlock(&routelock);
40   return -ENOENT;
41 }
```

even further, given that people repairing large dangerous machines have long used a mechanical reference-counting technique implemented via padlocks. Before entering the machine, each worker locks a padlock onto the machine's on/off switch, thus preventing the machine from being powered on while that worker is inside. Reference counting is thus an excellent time-honored candidate for a concurrent implementation of Pre-BSD routing.

To that end, Listing 9.2 shows data structures and the `route_lookup()` function and Listing 9.3 shows the `route_add()` and `route_del()` functions (all at `route_refcnt.c`). Since these algorithms are quite similar to the sequential algorithm shown in Listing 9.1, only the differences will be discussed.

Starting with Listing 9.2, line 2 adds the actual reference counter, line 6 adds a `->re_freed` use-after-free check field, line 9 adds the `routelock` that will be used to synchronize concurrent updates, and lines 11–15 add

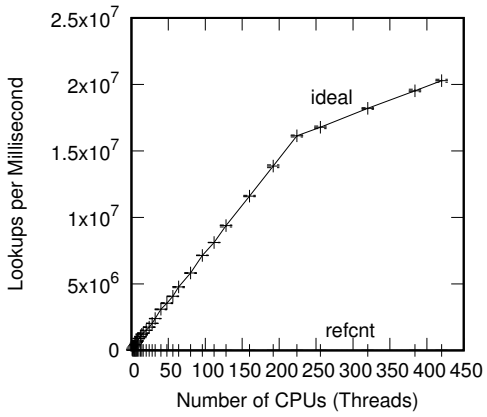


Figure 9.2: Pre-BSD Routing Table Protected by Reference Counting

`re_free()`, which sets `->re_freed`, enabling `route_lookup()` to check for use-after-free bugs. In `route_lookup()` itself, lines 29–30 release the reference count of the prior element and free it if the count becomes zero, and lines 34–42 acquire a reference on the new element, with lines 35 and 36 performing the use-after-free check.

Quick Quiz 9.1: Why bother with a use-after-free check? ■

In Listing 9.3, lines 11, 15, 24, 32, and 39 introduce locking to synchronize concurrent updates. Line 13 initializes the `->re_freed` use-after-free-check field, and finally lines 33–34 invoke `re_free()` if the new value of the reference count is zero.

Quick Quiz 9.2: Why doesn't `route_del()` in Listing 9.3 use reference counts to protect the traversal to the element to be freed? ■

Figure 9.2 shows the performance and scalability of reference counting on a read-only workload with a ten-element list running on an eight-socket 28-core-per-socket hyperthreaded 2.1 GHz x86 system with a total of 448 hardware threads (`hps.2019.12.02a/lscpu.hps`). The "ideal" trace was generated by running the sequential code shown in Listing 9.1, which works only because this is a read-only workload. The reference-counting performance is abysmal and its scalability even more so, with the "refcnt" trace indistinguishable from the x-axis. This should be no surprise in view of Chapter 3: The reference-count acquisitions and releases have added frequent shared-memory writes to an otherwise read-only workload, thus incurring severe retribution from the laws of physics. As well it should, given that all the wishful thinking in the world is not going to increase the speed

www.dbooks.org

of light or decrease the size of the atoms used in modern digital electronics.

Quick Quiz 9.3: Why the break in the "ideal" line at 224 CPUs in Figure 9.2? Shouldn't it be a straight line? ■

Quick Quiz 9.4: Shouldn't the refcnt trace in Figure 9.2 be at least a little bit off of the x-axis??? ■

But it gets worse.

Running multiple updater threads repeatedly invoking `route_add()` and `route_del()` will quickly encounter the `abort()` statement on line 36 of Listing 9.2, which indicates a use-after-free bug. This in turn means that the reference counts are not only profoundly degrading scalability and performance, but also failing to provide the needed protection.

One sequence of events leading to the use-after-free bug is as follows, given the list shown in Figure 9.1:

1. Thread A looks up address 42, reaching line 32 of `route_lookup()` in Listing 9.2. In other words, Thread A has a pointer to the first element, but has not yet acquired a reference to it.

2. Thread B invokes `route_del()` in Listing 9.3 to delete the route entry for address 42. It completes successfully, and because this entry's `->re_refcnt` field was equal to the value one, it invokes `re_free()` to set the `->re_freed` field and to free the entry.

3. Thread A continues execution of `route_lookup()`. Its `rep` pointer is non-`NULL`, but line 35 sees that its `->re_freed` field is non-zero, so line 36 invokes `abort()`.

The problem is that the reference count is located in the object to be protected, but that means that there is no protection during the instant in time when the reference count itself is being acquired! This is the reference-counting counterpart of a locking issue noted by Gamsa et al. [GKAS99]. One could imagine using a global lock or reference count to protect the per-route-entry reference-count acquisition, but this would result in severe contention issues. Although algorithms exist that allow safe reference-count acquisition in a concurrent environment [Val95], they are not only extremely complex and error-prone [MS95], but also provide terrible performance and scalability [HMBW07].

In short, concurrency has most definitely reduced the usefulness of reference counting!

Quick Quiz 9.5: If concurrency has "most definitely reduced the usefulness of reference counting", why are there so many reference counters in the Linux kernel? ■

That said, sometimes it is necessary to look at a problem in an entirely different way in order to successfully solve it. The next section describes what could be thought of as an inside-out reference count that provides decent performance and scalability.

9.3 Hazard Pointers

> If in doubt, turn it inside out.
>
> *Zara Carpenter*

One way of avoiding problems with concurrent reference counting is to implement the reference counters inside out, that is, rather than incrementing an integer stored in the data element, instead store a pointer to that data element in per-CPU (or per-thread) lists. Each element of these lists is called a *hazard pointer* [Mic04].[2] The value of a given data element's "virtual reference counter" can then be obtained by counting the number of hazard pointers referencing that element. Therefore, if that element has been rendered inaccessible to readers, and there are no longer any hazard pointers referencing it, that element may safely be freed.

Of course, this means that hazard-pointer acquisition must be carried out quite carefully in order to avoid destructive races with concurrent deletion. One implementation is shown in Listing 9.4, which shows `hp_try_record()` on lines 1–16, `hp_record()` on lines 18–27, and `hp_clear()` on lines 29–33 (`hazptr.h`).

The `hp_try_record()` macro on line 16 is simply a casting wrapper for the `_h_t_r_impl()` function, which attempts to store the pointer referenced by `p` into the hazard pointer referenced by `hp`. If successful, it returns the value of the stored pointer. If it fails due to that pointer being `NULL`, it returns `NULL`. Finally, if it fails due to racing with an update, it returns a special `HAZPTR_POISON` token.

Quick Quiz 9.6: Given that papers on hazard pointers use the bottom bits of each pointer to mark deleted elements, what is up with `HAZPTR_POISON`? ■

Line 6 reads the pointer to the object to be protected. If line 8 finds that this pointer was either `NULL` or the special `HAZPTR_POISON` deleted-object token, it returns

[2] Also independently invented by others [HLM02].

Listing 9.4: Hazard-Pointer Recording and Clearing

```
1  static inline void *_h_t_r_impl(void **p,
2                                  hazard_pointer *hp)
3  {
4    void *tmp;
5
6    tmp = READ_ONCE(*p);
7    if (!tmp || tmp == (void *)HAZPTR_POISON)
8      return tmp;
9    WRITE_ONCE(hp->p, tmp);
10   smp_mb();
11   if (tmp == READ_ONCE(*p))
12     return tmp;
13   return (void *)HAZPTR_POISON;
14 }
15
16 #define hp_try_record(p, hp) _h_t_r_impl((void **)(p), hp)
17
18 static inline void *hp_record(void **p,
19                               hazard_pointer *hp)
20 {
21   void *tmp;
22
23   do {
24     tmp = hp_try_record(*p, hp);
25   } while (tmp == (void *)HAZPTR_POISON);
26   return tmp;
27 }
28
29 static inline void hp_clear(hazard_pointer *hp)
30 {
31   smp_mb();
32   WRITE_ONCE(hp->p, NULL);
33 }
```

the pointer's value to inform the caller of the failure. Otherwise, line 9 stores the pointer into the specified hazard pointer, and line 10 forces full ordering of that store with the reload of the original pointer on line 11. (See Chapter 15 for more information on memory ordering.) If the value of the original pointer has not changed, then the hazard pointer protects the pointed-to object, and in that case, line 12 returns a pointer to that object, which also indicates success to the caller. Otherwise, if the pointer changed between the two `READ_ONCE()` invocations, line 13 indicates failure.

Quick Quiz 9.7: Why does `hp_try_record()` in Listing 9.4 take a double indirection to the data element? Why not `void *` instead of `void **`? ■

The `hp_record()` function is quite straightforward: It repeatedly invokes `hp_try_record()` until the return value is something other than `HAZPTR_POISON`.

Quick Quiz 9.8: Why bother with `hp_try_record()`? Wouldn't it be easier to just use the failure-immune `hp_record()` function? ■

The `hp_clear()` function is even more straightforward, with an `smp_mb()` to force full ordering between the caller's uses of the object protected by the hazard pointer and the setting of the hazard pointer to `NULL`.

Once a hazard-pointer-protected object has been removed from its linked data structure, so that it is now inaccessible to future hazard-pointer readers, it is passed to `hazptr_free_later()`, which is shown on lines 48–56 of Listing 9.5 (`hazptr.c`). Lines 50 and 51 enqueue the object on a per-thread list `rlist` and line 52 counts the object in `rcount`. If line 53 sees that a sufficiently large number of objects are now queued, line 54 invokes `hazptr_scan()` to attempt to free some of them.

The `hazptr_scan()` function is shown on lines 6–46 of the listing. This function relies on a fixed maximum number of threads (`NR_THREADS`) and a fixed maximum number of hazard pointers per thread (`K`), which allows a fixed-size array of hazard pointers to be used. Because any thread might need to scan the hazard pointers, each thread maintains its own array, which is referenced by the per-thread variable `gplist`. If line 14 determines that this thread has not yet allocated its `gplist`, lines 15–18 carry out the allocation. The memory barrier on line 20 ensures that all threads see the removal of all objects by this thread before lines 22–28 scan all of the hazard pointers, accumulating non-NULL pointers into the `plist` array and counting them in `psize`. The memory barrier on line 29 ensures that the reads of the hazard pointers happen before any objects are freed. Line 30 then sorts this array to enable use of binary search below.

Lines 31 and 32 remove all elements from this thread's list of to-be-freed objects, placing them on the local `tmplist` and line 33 zeroes the count. Each pass through the loop spanning lines 34–45 processes each of the to-be-freed objects. Lines 35 and 36 remove the first object from `tmplist`, and if lines 37 and 38 determine that there is a hazard pointer protecting this object, lines 39–41 place it back onto `rlist`. Otherwise, line 43 frees the object.

The Pre-BSD routing example can use hazard pointers as shown in Listing 9.6 for data structures and `route_lookup()`, and in Listing 9.7 for `route_add()` and `route_del()` (`route_hazptr.c`). As with reference counting, the hazard-pointers implementation is quite similar to the sequential algorithm shown in Listing 9.1 on page 126, so only differences will be discussed.

Starting with Listing 9.6, line 2 shows the `->hh` field used to queue objects pending hazard-pointer free, line 6 shows the `->re_freed` field used to detect use-after-free bugs, and line 21 invokes `hp_try_record()` to attempt to acquire a hazard pointer. If the return value is `NULL`, line 23 returns a not-found indication to the caller. If the

www.dbooks.org

Listing 9.5: Hazard-Pointer Scanning and Freeing

```
1  int compare(const void *a, const void *b)
2  {
3    return ( *(hazptr_head_t **)a - *(hazptr_head_t **)b );
4  }
5
6  void hazptr_scan()
7  {
8    hazptr_head_t *cur;
9    int i;
10   hazptr_head_t *tmplist;
11   hazptr_head_t **plist = gplist;
12   unsigned long psize;
13
14   if (plist == NULL) {
15     psize = sizeof(hazptr_head_t *) * K * NR_THREADS;
16     plist = (hazptr_head_t **)malloc(psize);
17     BUG_ON(!plist);
18     gplist = plist;
19   }
20   smp_mb();
21   psize = 0;
22   for (i = 0; i < H; i++) {
23     uintptr_t hp = (uintptr_t)READ_ONCE(HP[i].p);
24
25     if (!hp)
26       continue;
27     plist[psize++] = (hazptr_head_t *)(hp & ~0x1UL);
28   }
29   smp_mb();
30   qsort(plist, psize, sizeof(hazptr_head_t *), compare);
31   tmplist = rlist;
32   rlist = NULL;
33   rcount = 0;
34   while (tmplist != NULL) {
35     cur = tmplist;
36     tmplist = tmplist->next;
37     if (bsearch(&cur, plist, psize,
38                 sizeof(hazptr_head_t *), compare)) {
39       cur->next = rlist;
40       rlist = cur;
41       rcount++;
42     } else {
43       hazptr_free(cur);
44     }
45   }
46 }
47
48 void hazptr_free_later(hazptr_head_t *n)
49 {
50   n->next = rlist;
51   rlist = n;
52   rcount++;
53   if (rcount >= R) {
54     hazptr_scan();
55   }
56 }
```

Listing 9.6: Hazard-Pointer Pre-BSD Routing Table Lookup

```
1  struct route_entry {
2    struct hazptr_head hh;
3    struct route_entry *re_next;
4    unsigned long addr;
5    unsigned long iface;
6    int re_freed;
7  };
8  struct route_entry route_list;
9  DEFINE_SPINLOCK(routelock);
10 hazard_pointer __thread *my_hazptr;
11
12 unsigned long route_lookup(unsigned long addr)
13 {
14   int offset = 0;
15   struct route_entry *rep;
16   struct route_entry **repp;
17
18 retry:
19   repp = &route_list.re_next;
20   do {
21     rep = hp_try_record(repp, &my_hazptr[offset]);
22     if (!rep)
23       return ULONG_MAX;
24     if ((uintptr_t)rep == HAZPTR_POISON)
25       goto retry;
26     repp = &rep->re_next;
27   } while (rep->addr != addr);
28   if (READ_ONCE(rep->re_freed))
29     abort();
30   return rep->iface;
31 }
```

call to `hp_try_record()` raced with deletion, line 25 branches back to line 18's `retry` to re-traverse the list from the beginning. The `do`–`while` loop falls through when the desired element is located, but if this element has already been freed, line 29 terminates the program. Otherwise, the element's `->iface` field is returned to the caller.

Note that line 21 invokes `hp_try_record()` rather than the easier-to-use `hp_record()`, restarting the full search upon `hp_try_record()` failure. And such restarting is absolutely required for correctness. To see this, consider a hazard-pointer-protected linked list containing elements A, B, and C that is subjected to the following sequence of events:

1. Thread 0 stores a hazard pointer to element B (having presumably traversed to element B from element A).
2. Thread 1 removes element B from the list, which sets the pointer from element B to element C to the special `HAZPTR_POISON` value in order to mark the deletion. Because Thread 0 has a hazard pointer to element B, it cannot yet be freed.
3. Thread 1 removes element C from the list. Because there are no hazard pointers referencing element C, it is immediately freed.

4. Thread 0 attempts to acquire a hazard pointer to now-removed element B's successor, but `hp_try_record()` returns the `HAZPTR_POISON` value, forcing the caller to restart its traversal from the beginning of the list.

Which is a very good thing, because B's successor is the now-freed element C, which means that Thread 0's subsequent accesses might have resulted in arbitrarily horrible memory corruption, especially if the memory for element C had since been re-allocated for some other purpose. Therefore, hazard-pointer readers must typically restart the full traversal in the face of a concurrent deletion. Often the restart must go back to some global (and thus immortal) pointer, but it is sometimes possible to restart at some intermediate location if that location is guaranteed to still be live, for example, due to the current thread holding a lock, a reference count, etc.

Quick Quiz 9.9: Readers must "typically" restart? What are some exceptions? ■

Because algorithms using hazard pointers might be restarted at any step of their traversal through the linked data structure, such algorithms must typically take care to avoid making any changes to the data structure until after they have acquired all the hazard pointers that are required for the update in question.

Quick Quiz 9.10: But don't these restrictions on hazard pointers also apply to other forms of reference counting? ■

These hazard-pointer restrictions result in great benefits to readers, courtesy of the fact that the hazard pointers are stored local to each CPU or thread, which in turn allows traversals to be carried out without any writes to the data structures being traversed. Referring back to Figure 5.8 on page 71, hazard pointers enable the CPU caches to do resource replication, which in turn allows weakening of the parallel-access-control mechanism, thus boosting performance and scalability.

Another advantage of restarting hazard pointers traversals is a reduction in minimal memory footprint: Any object not currently referenced by some hazard pointer may be immediately freed. In contrast, Section 9.5 will discuss a mechanism that avoids read-side retries (and minimizes read-side overhead), but which can result in a much larger memory footprint.

The `route_add()` and `route_del()` functions are shown in Listing 9.7. Line 10 initializes `->re_freed`, line 31 poisons the `->re_next` field of the newly removed object, and line 33 passes that object to the `hazptr_free_later()` function, which will free that object once it is safe to do so. The spinlocks work the same as in Listing 9.3.

Listing 9.7: Hazard-Pointer Pre-BSD Routing Table Add/Delete

```
1  int route_add(unsigned long addr, unsigned long interface)
2  {
3    struct route_entry *rep;
4
5    rep = malloc(sizeof(*rep));
6    if (!rep)
7      return -ENOMEM;
8    rep->addr = addr;
9    rep->iface = interface;
10   rep->re_freed = 0;
11   spin_lock(&routelock);
12   rep->re_next = route_list.re_next;
13   route_list.re_next = rep;
14   spin_unlock(&routelock);
15   return 0;
16 }
17
18 int route_del(unsigned long addr)
19 {
20   struct route_entry *rep;
21   struct route_entry **repp;
22
23   spin_lock(&routelock);
24   repp = &route_list.re_next;
25   for (;;) {
26     rep = *repp;
27     if (rep == NULL)
28       break;
29     if (rep->addr == addr) {
30       *repp = rep->re_next;
31       rep->re_next = (struct route_entry *)HAZPTR_POISON;
32       spin_unlock(&routelock);
33       hazptr_free_later(&rep->hh);
34       return 0;
35     }
36     repp = &rep->re_next;
37   }
38   spin_unlock(&routelock);
39   return -ENOENT;
40 }
```

Figure 9.3 shows the hazard-pointers-protected Pre-BSD routing algorithm's performance on the same read-only workload as for Figure 9.2. Although hazard pointers scale far better than does reference counting, hazard pointers still require readers to do writes to shared memory (albeit with much improved locality of reference), and also require a full memory barrier and retry check for each object traversed. Therefore, hazard-pointers performance is still far short of ideal. On the other hand, unlike naive approaches to concurrent reference-counting, hazard pointers not only operate correctly for workloads involving concurrent updates, but also exhibit excellent scalability. Additional performance comparisons with other mechanisms may be found in Chapter 10 and in other publications [HMBW07, McK13, Mic04].

www.dbooks.org

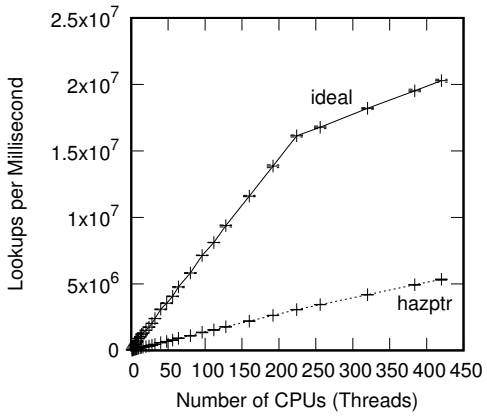


Figure 9.3: Pre-BSD Routing Table Protected by Hazard Pointers

Quick Quiz 9.11: Figure 9.3 shows no sign of hyperthread-induced flattening at 224 threads. Why is that? ■

Quick Quiz 9.12: The paper "Structured Deferral: Synchronization via Procrastination" [McK13] shows that hazard pointers have near-ideal performance. Whatever happened in Figure 9.3??? ■

The next section attempts to improve on hazard pointers by using sequence locks, which avoid both read-side writes and per-object memory barriers.

9.4 Sequence Locks

> It'll be just like starting over.
>
> *John Lennon*

The published sequence-lock record [Eas71, Lam77] extends back as far as that of reader-writer locking, but sequence locks nevertheless remain in relative obscurity. Sequence locks are used in the Linux kernel for read-mostly data that must be seen in a consistent state by readers. However, unlike reader-writer locking, readers do not exclude writers. Instead, like hazard pointers, sequence locks force readers to *retry* an operation if they detect activity from a concurrent writer. As can be seen from Figure 9.4, it is important to design code using sequence locks so that readers very rarely need to retry.

Quick Quiz 9.13: Why isn't this sequence-lock discussion in Chapter 7, you know, the one on *locking*? ■

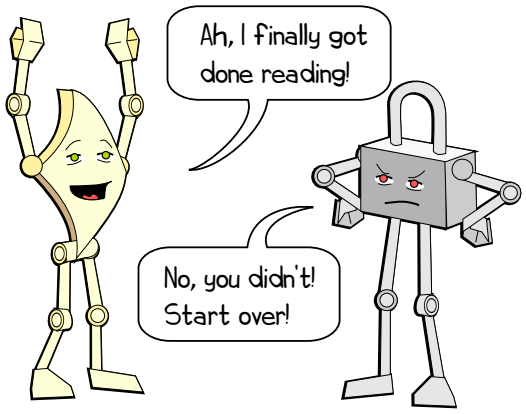


Figure 9.4: Reader And Uncooperative Sequence Lock

The key component of sequence locking is the sequence number, which has an even value in the absence of updaters and an odd value if there is an update in progress. Readers can then snapshot the value before and after each access. If either snapshot has an odd value, or if the two snapshots differ, there has been a concurrent update, and the reader must discard the results of the access and then retry it. Readers therefore use the `read_seqbegin()` and `read_seqretry()` functions shown in Listing 9.8 when accessing data protected by a sequence lock. Writers must increment the value before and after each update, and only one writer is permitted at a given time. Writers therefore use the `write_seqlock()` and `write_sequnlock()` functions shown in Listing 9.9 when updating data protected by a sequence lock.

As a result, sequence-lock-protected data can have an arbitrarily large number of concurrent readers, but only one writer at a time. Sequence locking is used in the Linux kernel to protect calibration quantities used for timekeeping. It is also used in pathname traversal to detect concurrent rename operations.

Listing 9.8: Sequence-Locking Reader

```
1 do {
2   seq = read_seqbegin(&test_seqlock);
3   /* read-side access. */
4 } while (read_seqretry(&test_seqlock, seq));
```

Listing 9.9: Sequence-Locking Writer

```
1 write_seqlock(&test_seqlock);
2 /* Update */
3 write_sequnlock(&test_seqlock);
```

Listing 9.10: Sequence-Locking Implementation

```
1  typedef struct {
2    unsigned long seq;
3    spinlock_t lock;
4  } seqlock_t;
5
6  static inline void seqlock_init(seqlock_t *slp)
7  {
8    slp->seq = 0;
9    spin_lock_init(&slp->lock);
10 }
11
12 static inline unsigned long read_seqbegin(seqlock_t *slp)
13 {
14   unsigned long s;
15
16   s = READ_ONCE(slp->seq);
17   smp_mb();
18   return s & ~0x1UL;
19 }
20
21 static inline int read_seqretry(seqlock_t *slp,
22                                 unsigned long oldseq)
23 {
24   unsigned long s;
25
26   smp_mb();
27   s = READ_ONCE(slp->seq);
28   return s != oldseq;
29 }
30
31 static inline void write_seqlock(seqlock_t *slp)
32 {
33   spin_lock(&slp->lock);
34   ++slp->seq;
35   smp_mb();
36 }
37
38 static inline void write_sequnlock(seqlock_t *slp)
39 {
40   smp_mb();
41   ++slp->seq;
42   spin_unlock(&slp->lock);
43 }
```

A simple implementation of sequence locks is shown in Listing 9.10 (`seqlock.h`). The `seqlock_t` data structure is shown on lines 1–4, and contains the sequence number along with a lock to serialize writers. Lines 6–10 show `seqlock_init()`, which, as the name indicates, initializes a `seqlock_t`.

Lines 12–19 show `read_seqbegin()`, which begins a sequence-lock read-side critical section. Line 16 takes a snapshot of the sequence counter, and line 17 orders this snapshot operation before the caller's critical section. Finally, line 18 returns the value of the snapshot (with the least-significant bit cleared), which the caller will pass to a later call to `read_seqretry()`.

Quick Quiz 9.14: Why not have `read_seqbegin()` in Listing 9.10 check for the low-order bit being set, and retry internally, rather than allowing a doomed read to start? ■

Lines 21–29 show `read_seqretry()`, which returns true if there was at least one writer since the time of the corresponding call to `read_seqbegin()`. Line 26 orders the caller's prior critical section before line 27's fetch of the new snapshot of the sequence counter. Line 28 checks whether the sequence counter has changed, in other words, whether there has been at least one writer, and returns true if so.

Quick Quiz 9.15: Why is the `smp_mb()` on line 26 of Listing 9.10 needed? ■

Quick Quiz 9.16: Can't weaker memory barriers be used in the code in Listing 9.10? ■

Quick Quiz 9.17: What prevents sequence-locking updaters from starving readers? ■

Lines 31–36 show `write_seqlock()`, which simply acquires the lock, increments the sequence number, and executes a memory barrier to ensure that this increment is ordered before the caller's critical section. Lines 38–43 show `write_sequnlock()`, which executes a memory barrier to ensure that the caller's critical section is ordered before the increment of the sequence number on line 41, then releases the lock.

Quick Quiz 9.18: What if something else serializes writers, so that the lock is not needed? ■

Quick Quiz 9.19: Why isn't `seq` on line 2 of Listing 9.10 `unsigned` rather than `unsigned long`? After all, if `unsigned` is good enough for the Linux kernel, shouldn't it be good enough for everyone? ■

So what happens when sequence locking is applied to the Pre-BSD routing table? Listing 9.11 shows the data structures and `route_lookup()`, and Listing 9.12 shows `route_add()` and `route_del()` (`route_seqlock.c`). This implementation is once again similar to its counterparts in earlier sections, so only the differences will be highlighted.

In Listing 9.11, line 5 adds `->re_freed`, which is checked on lines 29 and 30. Line 8 adds a sequence lock, which is used by `route_lookup()` on lines 18, 23, and 32, with lines 24 and 33 branching back to the `retry` label on line 17. The effect is to retry any lookup that runs concurrently with an update.

In Listing 9.12, lines 11, 14, 23, 31, and 39 acquire and release the sequence lock, while lines 10 and 33 handle `->re_freed`. This implementation is therefore quite straightforward.

www.dbooks.org

Listing 9.11: Sequence-Locked Pre-BSD Routing Table Lookup (BUGGY!!!)

```
1 struct route_entry {
2   struct route_entry *re_next;
3   unsigned long addr;
4   unsigned long iface;
5   int re_freed;
6 };
7 struct route_entry route_list;
8 DEFINE_SEQ_LOCK(sl);
9
10 unsigned long route_lookup(unsigned long addr)
11 {
12   struct route_entry *rep;
13   struct route_entry **repp;
14   unsigned long ret;
15   unsigned long s;
16
17 retry:
18   s = read_seqbegin(&sl);
19   repp = &route_list.re_next;
20   do {
21     rep = READ_ONCE(*repp);
22     if (rep == NULL) {
23       if (read_seqretry(&sl, s))
24         goto retry;
25       return ULONG_MAX;
26     }
27     repp = &rep->re_next;
28   } while (rep->addr != addr);
29   if (READ_ONCE(rep->re_freed))
30     abort();
31   ret = rep->iface;
32   if (read_seqretry(&sl, s))
33     goto retry;
34   return ret;
35 }
```

Listing 9.12: Sequence-Locked Pre-BSD Routing Table Add/Delete (BUGGY!!!)

```
1 int route_add(unsigned long addr, unsigned long interface)
2 {
3   struct route_entry *rep;
4
5   rep = malloc(sizeof(*rep));
6   if (!rep)
7     return -ENOMEM;
8   rep->addr = addr;
9   rep->iface = interface;
10  rep->re_freed = 0;
11  write_seqlock(&sl);
12  rep->re_next = route_list.re_next;
13  route_list.re_next = rep;
14  write_sequnlock(&sl);
15  return 0;
16 }
17
18 int route_del(unsigned long addr)
19 {
20   struct route_entry *rep;
21   struct route_entry **repp;
22
23   write_seqlock(&sl);
24   repp = &route_list.re_next;
25   for (;;) {
26     rep = *repp;
27     if (rep == NULL)
28       break;
29     if (rep->addr == addr) {
30       *repp = rep->re_next;
31       write_sequnlock(&sl);
32       smp_mb();
33       rep->re_freed = 1;
34       free(rep);
35       return 0;
36     }
37     repp = &rep->re_next;
38   }
39   write_sequnlock(&sl);
40   return -ENOENT;
41 }
```

It also performs better on the read-only workload, as can be seen in Figure 9.5, though its performance is still far from ideal. Worse yet, it suffers use-after-free failures. The problem is that the reader might encounter a segmentation violation due to accessing an already-freed structure before `read_seqretry()` has a chance to warn of the concurrent update.

Quick Quiz 9.20: Can this bug be fixed? In other words, can you use sequence locks as the *only* synchronization mechanism protecting a linked list supporting concurrent addition, deletion, and lookup? ■

Both the read-side and write-side critical sections of a sequence lock can be thought of as transactions, and sequence locking therefore can be thought of as a limited form of transactional memory, which will be discussed in Section 17.2. The limitations of sequence locking are: (1) Sequence locking restricts updates and (2) sequence locking does not permit traversal of pointers to objects that might be freed by updaters. These limitations are of course overcome by transactional memory, but can also be overcome by combining other synchronization primitives with sequence locking.

Sequence locks allow writers to defer readers, but not vice versa. This can result in unfairness and even starvation in writer-heavy workloads.[3] On the other hand, in the absence of writers, sequence-lock readers are reasonably fast and scale linearly. It is only human to want the best of both worlds: fast readers without the possibility of read-side failure, let alone starvation. In addition, it would also be nice to overcome sequence locking's limitations with pointers. The following section presents a synchronization mechanism with exactly these properties.

[3] Dmitry Vyukov describes one way to reduce (but, sadly, not eliminate) reader starvation: `http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/improved-lock-free-seqlock`.

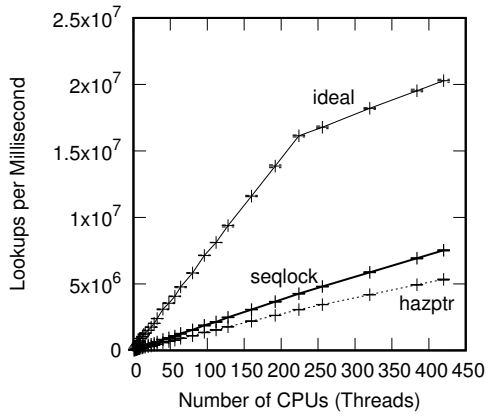


Figure 9.5: Pre-BSD Routing Table Protected by Sequence Locking

9.5 Read-Copy Update (RCU)

> "Free" is a *very* good price!
>
> *Tom Peterson*

All of the mechanisms discussed in the preceding sections used one of a number of approaches to defer specific actions until they may be carried out safely. The reference counters discussed in Section 9.2 use explicit counters to defer actions that could disturb readers, which results in read-side contention and thus poor scalability. The hazard pointers covered by Section 9.3 uses implicit counters in the guise of per-thread lists of pointer. This avoids read-side contention, but requires readers to do stores and conditional branches, as well as either full memory barriers in read-side primitives or real-time-unfriendly inter-processor interrupts in update-side primitives.[4] The sequence lock presented in Section 9.4 also avoids read-side contention, but does not protect pointer traversals and, like hazard pointers, requires either full memory barriers in read-side primitives, or inter-processor interrupts in update-side primitives. These schemes' shortcomings raise the question of whether it is possible to do better.

This section introduces *read-copy update* (RCU), which provides an API that allows readers to be associated with regions in the source code, rather than with expensive updates to frequently updated shared data. The remainder of this section examines RCU from a number of different perspectives. Section 9.5.1 provides the classic introduction to RCU, Section 9.5.2 covers fundamental RCU concepts, Section 9.5.3 presents the Linux-kernel API, Section 9.5.4 introduces some common uses of RCU, Section 9.5.5 covers recent work related to RCU, Section 9.5.6 provides some RCU exercises, and finally Section 9.7 discusses updates.

9.5.1 Introduction to RCU

The approaches discussed in the preceding sections have provided good scalability but decidedly non-ideal performance for the Pre-BSD routing table. Therefore, in the spirit of "only those who have gone too far know how far you can go",[5] we will go all the way, looking into algorithms in which concurrent readers execute the same sequence of assembly language instructions as would a single-threaded lookup, despite the presence of concurrent updates. Of course, this laudable goal might raise serious implementability questions, but we cannot possibly succeed if we don't even try!

9.5.1.1 Minimal Insertion and Deletion

To minimize implementability concerns, we focus on a minimal data structure, which consists of a single global pointer that is either `NULL` or references a single structure. Minimal though it might be, this data structure is heavily used in production [RH18]. A classic approach for insertion is shown in Figure 9.6, which shows four states with time advancing from top to bottom. The first row shows the initial state, with `gptr` equal to `NULL`. In the second row, we have allocated a structure which is uninitialized, as indicated by the question marks. In the third row, we have initialized the structure. Finally, in the fourth and final row, we have updated `gptr` to reference the newly allocated and initialized element.

We might hope that this assignment to `gptr` could use a simple C-language assignment statement. Unfortunately, Section 4.3.4.1 dashes these hopes. Therefore, the updater cannot use a simple C-language assignment, but must instead use `smp_store_release()` as shown in the figure, or, as will be seen, `rcu_assign_pointer()`.

Similarly, one might hope that readers could use a single C-language assignment to fetch the value of `gptr`, and be guaranteed to either get the old value of `NULL` or to get the newly installed pointer, but either way see a valid result. Unfortunately, Section 4.3.4.1 dashes these hopes

[4] In some important special cases, this extra work can be avoided by using link counting as exemplified by the UnboundedQueue and ConcurrentHashMap data structures implemented in Folly open-source library (`https://github.com/facebook/folly`).

[5] With apologies to T. S. Eliot.

www.dbooks.org

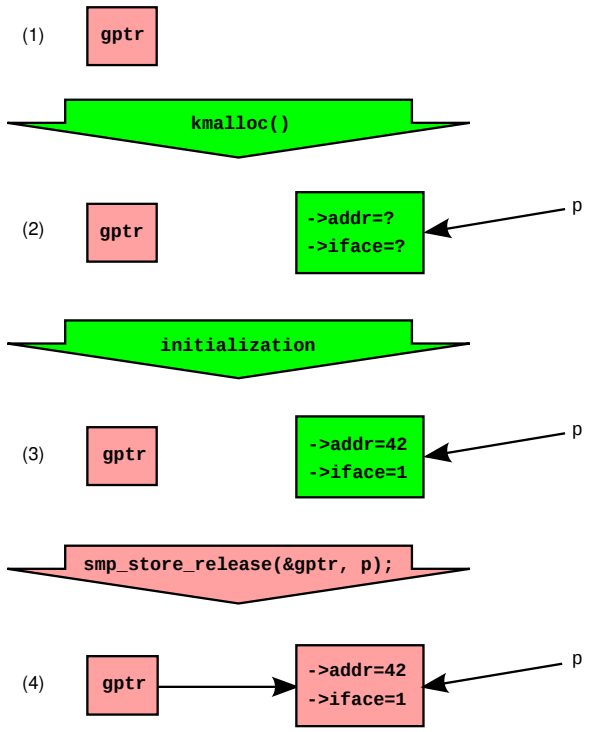


Figure 9.6: Insertion With Concurrent Readers

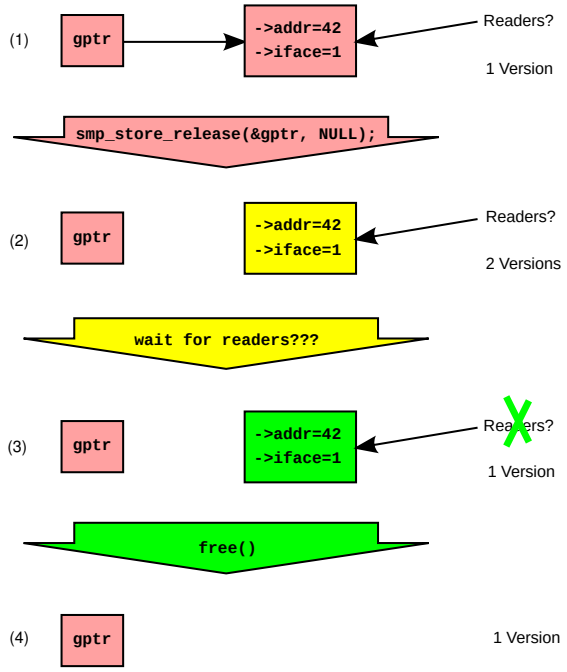


Figure 9.7: Deletion With Concurrent Readers

as well. To obtain this guarantee, readers must instead use `READ_ONCE()`, or, as will be seen, `rcu_dereference()`. However, on most modern computer systems, each of these read-side primitives can be implemented with a single load instruction, exactly the instruction that would normally be used in single-threaded code.

Reviewing Figure 9.6 from the viewpoint of readers, in the first three states all readers see `gptr` having the value `NULL`. Upon entering the fourth state, some readers might see `gptr` still having the value `NULL` while others might see it referencing the newly inserted element, but after some time, all readers will see this new element. At all times, all readers will see `gptr` as containing a valid pointer. Therefore, it really is possible to add new data to linked data structures while allowing concurrent readers to execute the same sequence of machine instructions that is normally used in single-threaded code. This no-cost approach to concurrent reading provides excellent performance and scalability, and also is eminently suitable for real-time use.

Insertion is of course quite useful, but sooner or later, it will also be necessary to delete data. As can be seen in Figure 9.7, the first step is easy. Again taking the lessons from Section 4.3.4.1 to heart, `smp_store_release()` is used to `NULL` the pointer, thus moving from the first row to the second in the figure. At this point, pre-existing readers see the old structure with `->addr` of 42 and `->iface` of 1, but new readers will see a `NULL` pointer, that is, concurrent readers can disagree on the state, as indicated by the "2 Versions" in the figure.

Quick Quiz 9.21: Why does Figure 9.7 use `smp_store_release()` given that it is storing a `NULL` pointer? Wouldn't `WRITE_ONCE()` work just as well in this case, given that there is no structure initialization to order against the store of the `NULL` pointer? ■

Quick Quiz 9.22: Readers running concurrently each other and with the procedure outlined in Figure 9.7 can disagree on the value of `gptr`. Isn't that just a wee bit problematic??? ■

We get back to a single version simply by waiting for all the pre-existing readers to complete, as shown in row 3. At that point, all the pre-existing readers are done, and no later reader has a path to the old data item, so there can no longer be any readers referencing it. It may therefore be safely freed, as shown on row 4.

Thus, given a way to wait for pre-existing readers to complete, it is possible to both add data to and remove data

from a linked data structure, despite the readers executing the same sequence of machine instructions that would be appropriate for single-threaded execution. So perhaps going all the way was not too far after all!

But how can we tell when all of the pre-existing readers have in fact completed? This question is the topic of the next section.

9.5.1.2 Waiting for Readers

It is tempting to base reader waiting on reference counting, but Figure 5.1 in Chapter 5 shows that concurrent reference counting results in extreme overhead, as we already saw in Section 9.2. Hazard pointers profoundly reduce this overhead, but, as we saw in Section 9.3, not to zero. Nevertheless, many RCU implementations make very careful cache-local use of counters.

A second approach observes that memory synchronization is expensive, and therefore uses registers instead, namely each CPU's or thread's program counter (PC), thus imposing no overhead on readers, at least in the absence of concurrent updates. The updater polls each relevant PC, and if that PC is not within read-side code, then the corresponding CPU or thread is within a quiescent state, in turn signaling the completion of any reader that might have access to the newly removed data element. Once all CPU's or thread's PCs have been observed to be outside of any reader, the grace period has completed. Please note that this approach poses some serious challenges, including memory ordering, functions that are *sometimes* invoked from readers, and ever-exciting code-motion optimizations. Nevertheless, this approach is said to be used in production [Ash15].

A third approach is to simply wait for a fixed period of time that is long enough to comfortably exceed the lifetime of any reasonable reader [Jac93, Joh95]. This can work quite well in hard real-time systems [RLPB18], but in less exotic settings, Murphy says that it is critically important to be prepared even for unreasonably long-lived readers. To see this, consider the consequences of failing do so: A data item will be freed while the unreasonable reader is still referencing it, and that item might well be immediately reallocated, possibly even as a data item of some other type. The unreasonable reader and the unwitting reallocator would then be attempting to use the same memory for two very different purposes. The ensuing mess will at best be exceedingly difficult to debug.

A fourth approach is to wait forever, secure in the knowledge that doing so will accommodate even the most unreasonable reader. This approach is also called "leaking memory", and has a bad reputation due to the fact that memory leaks often require untimely and inconvenient reboots. Nevertheless, this is a viable strategy when the update rate and the uptime are both sharply bounded. For example, this approach could work well in a high-availability cluster where systems were periodically crashed in order to ensure that cluster really remained highly available.[6] Leaking the memory is also a viable strategy in environments having garbage collectors, in which case the garbage collector can be thought of as plugging the leak [KL80]. However, if your environment lacks a garbage collector, read on!

A fifth approach avoids the period crashes in favor of periodically "stopping the world", as exemplified by the traditional stop-the-world garbage collector. This approach was also heavily used during the decades before ubiquitous connectivity, when it was common practice to power systems off at the end of each working day. However, in today's always-connected always-on world, stopping the world can gravely degrade response times, which has been one motivation for the development of concurrent garbage collectors [BCR03]. Furthermore, although we need all pre-existing readers to complete, we do not need them all to complete at the same time.

This observation leads to the sixth approach, which is stopping one CPU or thread at a time. This approach has the advantage of not degrading reader response times at all, let alone gravely. Furthermore, numerous applications already have states (termed *quiescent states*) that can be reached only after all pre-existing readers are done. In transaction-processing systems, the time between a pair of successive transactions might be a quiescent state. In reactive systems, the state between a pair of successive events might be a quiescent state. Within non-preemptive operating-systems kernels, a context switch can be a quiescent state [MS98a]. Either way, once all CPUs and/or threads have passed through a quiescent state, the system is said to have completed a *grace period*, at which point all readers in existence at the start of that grace period are guaranteed to have completed. As a result, it is also guaranteed to be safe to free any removed data items that were removed prior to the start of that grace period.[7]

[6] The program that forces the periodic crashing is sometimes known as a "chaos monkey": `https://netflix.github.io/chaosmonkey/`. However, it might also be a mistake to neglect chaos caused by systems running for too long.

[7] It is possible to do much more with RCU than simply defer reclamation of memory, but deferred reclamation is RCU's most common use case, and is therefore an excellent place to start.

www.dbooks.org

Within a non-preemptive operating-system kernel, for context switch to be a valid quiescent state, readers must be prohibited from blocking while referencing a given instance data structure obtained via the gptr pointer shown in Figures 9.6 and 9.7. This no-blocking constraint is consistent with similar constraints on pure spinlocks, where a CPU is forbidden from blocking while holding a spinlock. Without this constraint, all CPUs might be consumed by threads spinning attempting to acquire a spinlock held by a blocked thread. The spinning threads will not relinquish their CPUs until they acquire the lock, but the thread holding the lock cannot possibly release it until one of the spinning threads relinquishes a CPU. This is a classic deadlock situation, and this deadlock is avoided by forbidding blocking while holding a spinlock.

Again, this same constraint is imposed on reader threads dereferencing gptr: such threads are not allowed to block until after they are done using the pointed-to data item. Returning to the second row of Figure 9.7, where the updater has just completed executing the smp_store_release(), imagine that CPU 0 executes a context switch. Because readers are not permitted to block while traversing the linked list, we are guaranteed that all prior readers that might have been running on CPU 0 will have completed. Extending this line of reasoning to the other CPUs, once each CPU has been observed executing a context switch, we are guaranteed that all prior readers have completed, and that there are no longer any reader threads referencing the newly removed data element. The updater can then safely free that data element, resulting in the state shown at the bottom of Figure 9.7.

This approach is termed *quiescent state based reclamation* (QSBR) [HMB06]. A QSBR schematic is shown in Figure 9.8, with time advancing from the top of the figure to the bottom. CPU 1 does the WRITE_ONCE() that removes the current data item (presumably having previously read the pointer value and availed itself of appropriate synchronization), then waits for readers. This wait operation results in an immediate context switch, which is a quiescent state (denoted by the pink circle), which in turn means that all prior reads on CPU 1 have completed. Next, CPU 2 does a context switch, so that all readers on CPUs 1 and 2 are now known to have completed. Finally, CPU 3 does a context switch. At this point, all readers throughout the entire system are known to have completed, so the grace period ends, permitting CPU 1 to free the old data item.

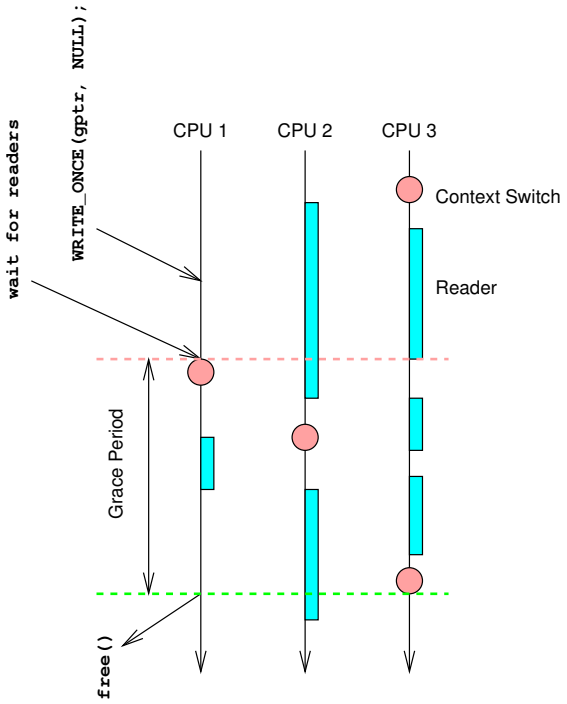


Figure 9.8: QSBR: Waiting for Pre-Existing Readers

Quick Quiz 9.23: In Figure 9.8, the last of CPU 3's readers that could possibly have access to the old data item ended before the grace period even started! So why would any anyone bother waiting until CPU 3's later context switch??? ■

9.5.1.3 Toy Implementation

Although production-quality QSBR implementations can be quite complex, a toy non-preemptive Linux-kernel implementation is exceedingly simple:

```
1 void synchronize_rcu(void)
2 {
3   int cpu;
4
5   for_each_online_cpu(cpu)
6     sched_setaffinity(current->pid, cpumask_of(cpu));
7 }
```

The for_each_online_cpu() primitive iterates over all CPUs, and the sched_setaffinity() function causes the current thread to execute on the specified CPU, which forces the destination CPU to execute a context switch. Therefore, once the for_each_online_cpu() has completed, each CPU has executed a context switch,

Listing 9.13: Insertion and Deletion With Concurrent Readers

```
1  struct route *gptr;
2
3  int access_route(int (*f)(struct route *rp))
4  {
5    int ret = -1;
6    struct route *rp;
7
8    rcu_read_lock();
9    rp = rcu_dereference(gptr);
10   if (rp)
11     ret = f(rp);
12   rcu_read_unlock();
13   return ret;
14 }
15
16 struct route *ins_route(struct route *rp)
17 {
18   struct route *old_rp;
19
20   spin_lock(&route_lock);
21   old_rp = gptr;
22   rcu_assign_pointer(gptr, rp);
23   spin_unlock(&route_lock);
24   return old_rp;
25 }
26
27 int del_route(void)
28 {
29   struct route *old_rp;
30
31   spin_lock(&route_lock);
32   old_rp = gptr;
33   RCU_INIT_POINTER(gptr, NULL);
34   spin_unlock(&route_lock);
35   synchronize_rcu();
36   free(old_rp);
37   return !!old_rp;
38 }
```

which in turn guarantees that all pre-existing reader threads have completed.

Please note that this approach is *not* production quality. Correct handling of a number of corner cases and the need for a number of powerful optimizations mean that production-quality implementations are quite complex. In addition, RCU implementations for preemptible environments require that readers actually do something, which in non-real-time Linux-kernel environments can be as simple as defining `rcu_read_lock()` and `rcu_read_unlock()` as `preempt_disable()` and `preempt_enable()`, respectively.[8] However, this simple non-preemptible approach is conceptually complete, and demonstrates that it really is possible to provide read-side synchronization at zero cost, even in the face of concurrent updates. In fact, Listing 9.13 shows how reading (`access_route()`), Figure 9.6's insertion (`ins_route()`) and Figure 9.7's deletion (`del_route()`) can be implemented. (A slightly more capable routing table is shown in Section 9.5.4.1.)

Quick Quiz 9.24: What is the point of `rcu_read_lock()` and `rcu_read_unlock()` in Listing 9.13? Why not just let the quiescent states speak for themselves? ■

Quick Quiz 9.25: What is the point of `rcu_dereference()`, `rcu_assign_pointer()` and `RCU_INIT_POINTER()` in Listing 9.13? Why not just use `READ_ONCE()`, `smp_store_release()`, and `WRITE_ONCE()`, respectively? ■

Referring back to Listing 9.13, note that `route_lock` is used to synchronize between concurrent updaters invoking `ins_route()` and `del_route()`. However, this lock is not acquired by readers invoking `access_route()`: Readers are instead protected by the QSBR techniques described in this section.

Note that `ins_route()` simply returns the old value of `gptr`, which Figure 9.6 assumed would always be `NULL`. This means that it is the caller's responsibility to figure out what to do with a non-`NULL` value, a task complicated by the fact that readers might still be referencing it for an indeterminate period of time. Callers might use one of the following approaches:

1. Use `synchronize_rcu()` to safely free the pointed-to structure. Although this approach is correct from an RCU perspective, it arguably has software-engineering leaky-API problems.
2. Trip an assertion if the returned pointer is non-`NULL`.
3. Pass the returned pointer to a later invocation of `ins_route()` to restore the earlier value.

In contrast, `del_route()` uses `synchronize_rcu()` and `free()` to safely free the newly deleted data item.

Quick Quiz 9.26: But what if the old structure needs to be freed, but the caller of `ins_route()` cannot block, perhaps due to performance considerations or perhaps because the caller is executing within an RCU read-side critical section? ■

This example shows one general approach to reading and updating RCU-protected data structures, however, there is quite a variety of use cases, several of which are covered in Section 9.5.4.

In summary, it is in fact possible to create concurrent linked data structures that can be traversed by readers executing the same sequence of machine instructions that would be executed by single-threaded readers. The next section summarizes RCU's high-level properties.

[8] Some toy RCU implementations that handle preempted read-side critical sections are shown in Appendix B.

www.dbooks.org

9.5.1.4 RCU Properties

A key RCU property is that reads need not wait for updates. This property enables RCU implementations to provide low-cost or even no-cost readers, resulting in low overhead and excellent scalability. This property also allows RCU readers and updaters to make useful concurrent forward progress. In contrast, conventional synchronization primitives must enforce strict mutual exclusion using expensive instructions, thus increasing overhead and degrading scalability, but also typically prohibiting readers and updaters from making useful concurrent forward progress.

Quick Quiz 9.27: Doesn't Section 9.4's seqlock also permit readers and updaters to make useful concurrent forward progress? ■

RCU delimits readers with `rcu_read_lock()` and `rcu_read_unlock()`, and ensures that each reader has a coherent view of each object (see Figure 9.7) by maintaining multiple versions of objects and using update-side primitives such as `synchronize_rcu()` to ensure that objects are not freed until after the completion of all readers that might be using them. RCU uses `rcu_assign_pointer()` and `rcu_dereference()` to provide efficient and scalable mechanisms for publishing and reading new versions of an object, respectively. These mechanisms distribute the work among read and update paths in such a way as to make read paths extremely fast, using replication and weakening optimizations in a manner similar to hazard pointers, but without the need for read-side retries. In some cases, including `CONFIG_PREEMPT=n` Linux kernels, RCU's read-side primitives have zero overhead.

But are these properties actually useful in practice? This question is taken up by the next section.

9.5.1.5 Practical Applicability

RCU has been used in the Linux kernel since October 2002 [Tor02]. Use of the RCU API has increased substantially since that time, as can be seen in Figure 9.9. In fact, code very similar to that in Listing 9.13 is used in the Linux kernel. RCU has enjoyed heavy use both prior to and since its acceptance in the Linux kernel, as discussed in Section 9.5.5.

It is therefore safe to say that RCU enjoys wide practical applicability.

The minimal example discussed in this section is a good introduction to RCU. However, effective use of RCU often

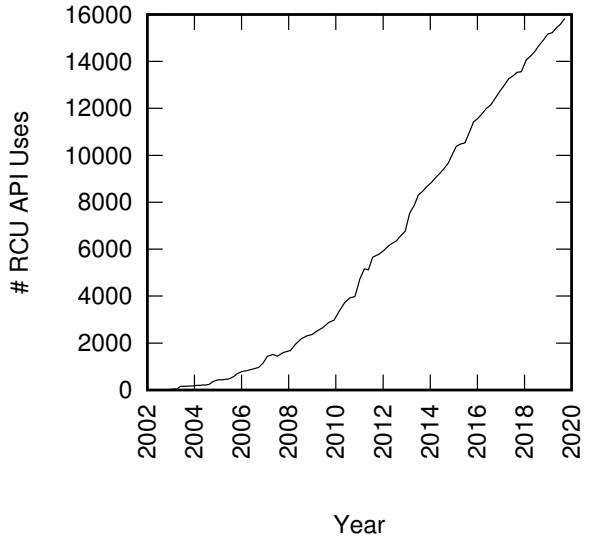


Figure 9.9: RCU Usage in the Linux Kernel

requires that you think differently about your problem. It is therefore useful to examine RCU's fundamentals, a task taken up by the following section.

9.5.2 RCU Fundamentals

This section re-examines the ground covered in the previous section, but independent of any particular example or use case. People who prefer to live their lives very close to the actual code may wish to skip the underlying fundamentals presented in this section.

RCU is made up of three fundamental mechanisms, the first being used for insertion, the second being used for deletion, and the third being used to allow readers to tolerate concurrent insertions and deletions. Section 9.5.2.1 describes the publish-subscribe mechanism used for insertion, Section 9.5.2.2 describes how waiting for pre-existing RCU readers enabled deletion, and Section 9.5.2.3 discusses how maintaining multiple versions of recently updated objects permits concurrent insertions and deletions. Finally, Section 9.5.2.4 summarizes RCU fundamentals.

9.5.2.1 Publish-Subscribe Mechanism

Because RCU readers are not excluded by RCU updaters, an RCU-protected data structure might change while a reader accesses it. The accessed data item might be moved, removed, or replaced. Because the data structure does not "hold still" for the reader, each reader's access can be thought of as subscribing to the current version of the

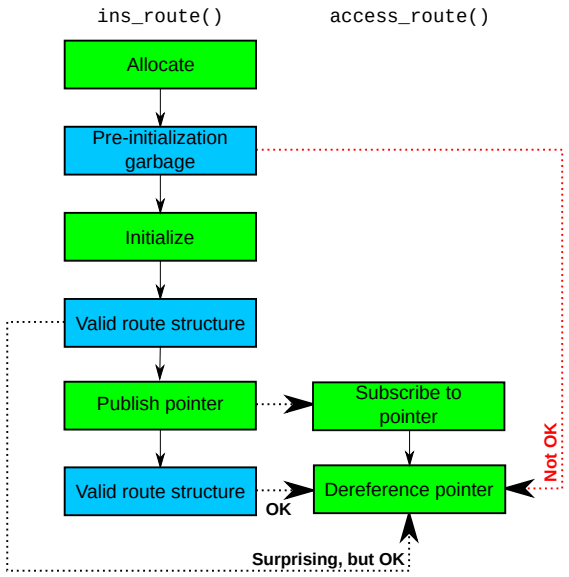


Figure 9.10: Publication/Subscription Constraints

RCU-protected data item. For their part, updaters can be thought of as publishing new versions.

Unfortunately, as laid out in Section 4.3.4.1 and reiterated in Section 9.5.1.1, it is unwise to use plain accesses for these publication and subscription operations. It is instead necessary to inform both the compiler and the CPU of the need for care, as can be seen from Figure 9.10, which illustrates interactions between concurrent executions of `ins_route()` (and its caller) and `read_gptr()` from Listing 9.13.

The `ins_route()` column from Figure 9.10 shows `ins_route()`'s caller allocating a new `route` structure, which then contains pre-initialization garbage. The caller then initializes the newly allocated structure, and then invokes `ins_route()` to publish a pointer to the new `route` structure. Publication does not affect the contents of the structure, which therefore remain valid after publication.

The `access_route()` column from this same figure shows the pointer being subscribed to and dereferenced. This dereference operation absolutely must see a valid `route` structure rather than pre-initialization garbage because referencing garbage could result in memory corruption, crashes, and hangs. As noted earlier, avoiding such garbage means that the publish and subscribe operations must inform both the compiler and the CPU of the need to maintain the needed ordering.

Publication is carried out by `rcu_assign_pointer()`, which ensures that `ins_route()`'s caller's initialization is ordered before the actual publication operation's store of the pointer. In addition, `rcu_assign_pointer()` must be atomic in the sense that concurrent readers see either the old value of the pointer or the new value of the pointer, but not some mash-up of these two values. These requirements are met by the C11 store-release operation, and in fact in the Linux kernel, `rcu_assign_pointer()` is defined in terms of `smp_store_release()`, which is similar to C11 store-release.

Note that if concurrent updates are required, some sort of synchronization mechanism will be required to mediate among multiple concurrent `rcu_assign_pointer()` calls on the same pointer. In the Linux kernel, locking is the mechanism of choice, but pretty much any synchronization mechanism may be used. An example of a particularly lightweight synchronization mechanism is Chapter 8's data ownership: If each pointer is owned by a particular thread, then that thread may execute `rcu_assign_pointer()` on that pointer with no additional synchronization overhead.

Quick Quiz 9.28: Wouldn't use of data ownership for RCU updaters mean that the updates could use exactly the same sequence of instructions as would the corresponding single-threaded code? ■

Subscription is carried out by `rcu_dereference()`, which orders the subscription operation's load from the pointer is before the dereference. Similar to `rcu_assign_pointer()`, `rcu_dereference()` must be atomic in the sense that the value loaded must be that from a single store, for example, the compiler must not tear the load.[9] Unfortunately, compiler support for `rcu_dereference()` is at best a work in progress [MWB+17, MRP+17, BM18]. In the meantime, the Linux kernel relies on volatile loads, the details of the various CPU architectures, coding restrictions [McK14c], and, on DEC Alpha [Cor02], a memory-barrier instruction. However, on other architectures, `rcu_dereference()` typically emits a single load instruction, just as would the equivalent single-threaded code. The coding restrictions are described in more detail in Section 15.3.2, however, the common case of field selection ("`->`") works quite well. Software that does not require the ultimate in read-side performance can instead use C11 acquire loads, which provide the needed ordering and more, albeit at a cost. It is hoped that lighter-weight compiler support for `rcu_dereference()` will appear in due course.

[9] That is, the compiler must not break the load into multiple smaller loads, as described under "load tearing" in Section 4.3.4.1.

www.dbooks.org

In short, use of `rcu_assign_pointer()` for publishing pointers and use of `rcu_dereference()` for subscribing to them successfully avoids the "Not OK" garbage loads depicted in Figure 9.10. These two primitives can therefore be used to add new data to linked structures without disrupting concurrent readers.

Quick Quiz 9.29: But suppose that updaters are adding and removing multiple data items from a linked list while a reader is iterating over that same list. Specifically, suppose that a list initially contains elements A, B, and C, and that an updater removes element A and then adds a new element D at the end of the list. The reader might well see {A, B, C, D}, when that sequence of elements never actually ever existed! In what alternate universe would that qualify as "not disrupting concurrent readers"??? ■

Adding data to a linked structure without disrupting readers is a good thing, as are the cases where this can be done with no added read-side cost compared to single-threaded readers. However, in most cases it is also necessary to remove data, and this is the subject of the next section.

9.5.2.2 Wait For Pre-Existing RCU Readers

In its most basic form, RCU is a way of waiting for things to finish. Of course, there are a great many other ways of waiting for things to finish, including reference counts, reader-writer locks, events, and so on. The great advantage of RCU is that it can wait for each of (say) 20,000 different things without having to explicitly track each and every one of them, and without having to worry about the performance degradation, scalability limitations, complex deadlock scenarios, and memory-leak hazards that are inherent in schemes using explicit tracking.

In RCU's case, each of the things waited on is called an *RCU read-side critical section*. As hinted at in Section 9.5.1.3, an RCU read-side critical section starts with an `rcu_read_lock()` primitive, and ends with a corresponding `rcu_read_unlock()` primitive. RCU read-side critical sections can be nested, and may contain pretty much any code, as long as that code does not contain a quiescent state, for example, within the Linux kernel, it is illegal to sleep within an RCU read-side critical section because a context switch is a quiescent state.[10] If you abide by these conventions, you can use RCU to wait for *any* pre-existing RCU read-side critical section to complete, and `synchronize_rcu()` uses indirect means to do the actual waiting [DMS+12, McK13].

[10] However, a special form of RCU called SRCU [McK06] does permit general sleeping in SRCU read-side critical sections.

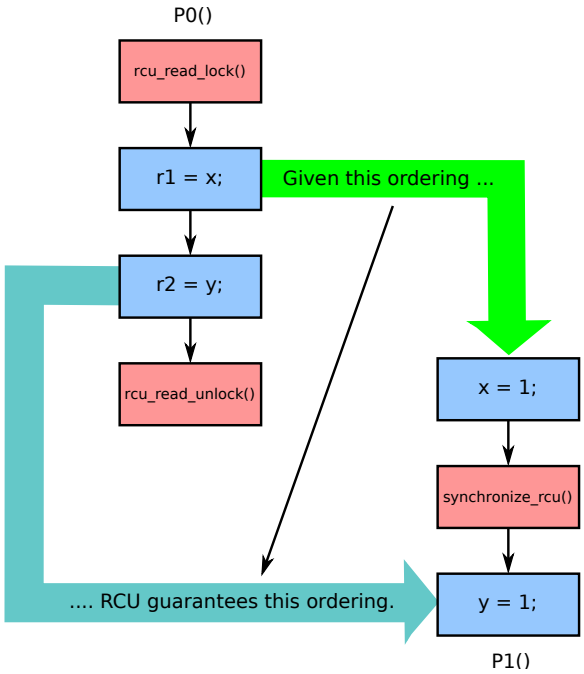


Figure 9.11: RCU Reader and Later Grace Period

The relationship between an RCU read-side critical section and a later RCU grace period is an if-then relationship, as illustrated by Figure 9.11. If any portion of a given critical section precedes the beginning of a given grace period, then RCU guarantees that all of that critical section will precede the end of that grace period. In the figure, `P0()`'s access to `x` precedes `P1()`'s access to this same variable, and thus also precedes the grace period generated by `P1()`'s call to `synchronize_rcu()`. It is therefore guaranteed that `P0()`'s access to `y` will precede `P1()`'s access. In this case, if `r1`'s final value is 0, then `r2`'s final value is guaranteed to also be 0.

Quick Quiz 9.30: What other final values of `r1` and `r2` are possible in Figure 9.11? ■

The relationship between an RCU read-side critical section and an earlier RCU grace period is also an if-then relationship, as illustrated by Figure 9.12. If any portion of a given critical section follows the end of a given grace period, then RCU guarantees that all of that critical section will follow the beginning of that grace period. In the figure, `P0()`'s access to `y` follows `P1()`'s access to this same variable, and thus follows the grace period

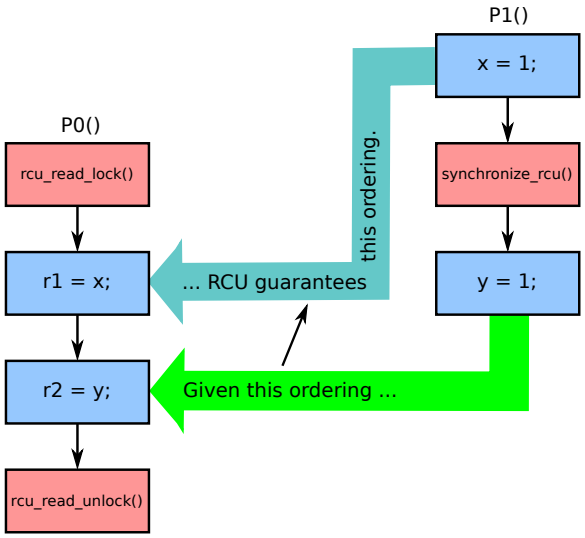


Figure 9.12: RCU Reader and Earlier Grace Period

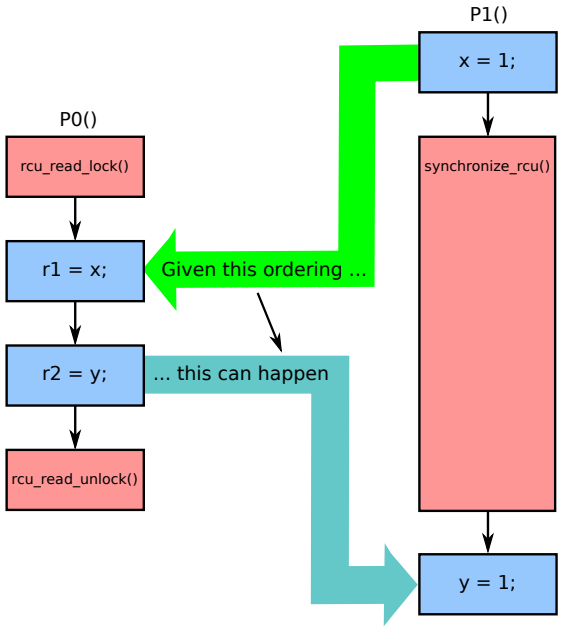


Figure 9.13: RCU Reader Within Grace Period

generated by `P1()`'s call to `synchronize_rcu()`. It is therefore guaranteed that `P0()`'s access to `x` will follow `P1()`'s access. In this case, if `r2`'s final value is 1, then `r1`'s final value is guaranteed to also be 1.

Quick Quiz 9.31: What would happen if the order of `P0()`'s two accesses was reversed in Figure 9.12? ■

Finally, as shown in Figure 9.13, an RCU read-side critical section can be completely overlapped by an RCU grace period. In this case, `r1`'s final value is 1 and `r2`'s final value is 0.

However, it cannot be the case that `r1`'s final value is 0 and `r2`'s final value is 1. This would mean that an RCU read-side critical section had completely overlapped a grace period, which is forbidden (or at the very least constitutes a bug in RCU). RCU's wait-for-readers guarantee therefore has two parts: (1) If any part of a given RCU read-side critical section precedes the beginning of a given grace period, then the entirety of that critical section precedes the end of that grace period. (2) If any part of a given RCU read-side critical section follows the end of a given grace period, then the entirety of that critical section follows the beginning of that grace period. This definition is sufficient for almost all RCU-based algorithms, but for those wanting more, simple executable formal models of RCU are available as part of Linux kernel v4.17 and later, as discussed in Section 12.3.2. In addition, RCU's ordering properties are examined in much greater detail in Section 15.4.2.

Quick Quiz 9.32: What would happen if `P0()`'s accesses in Figures 9.11–9.13 were stores? ■

Although RCU's wait-for-readers capability really is sometimes used to order the assignment of values to variables as shown in Figures 9.11–9.13, it is more frequently used to safely free data elements removed from a linked structure, as was done in Section 9.5.1. The general process is illustrated by the following pseudocode:

1. Make a change, for example, remove an element from a linked list.
2. Wait for all pre-existing RCU read-side critical sections to completely finish (for example, by using `synchronize_rcu()`).
3. Clean up, for example, free the element that was replaced above.

This more abstract procedure requires a more abstract diagram than Figures 9.11–9.13, which are specific to a particular litmus test. After all, and RCU implementation must work correctly regardless of the form of the RCU updates and the RCU read-side critical sections. Figure 9.14 fills this need, showing the four possible scenarios, with time advancing from top to bottom within each scenario. Within each scenario, an RCU reader is represented by

www.dbooks.org

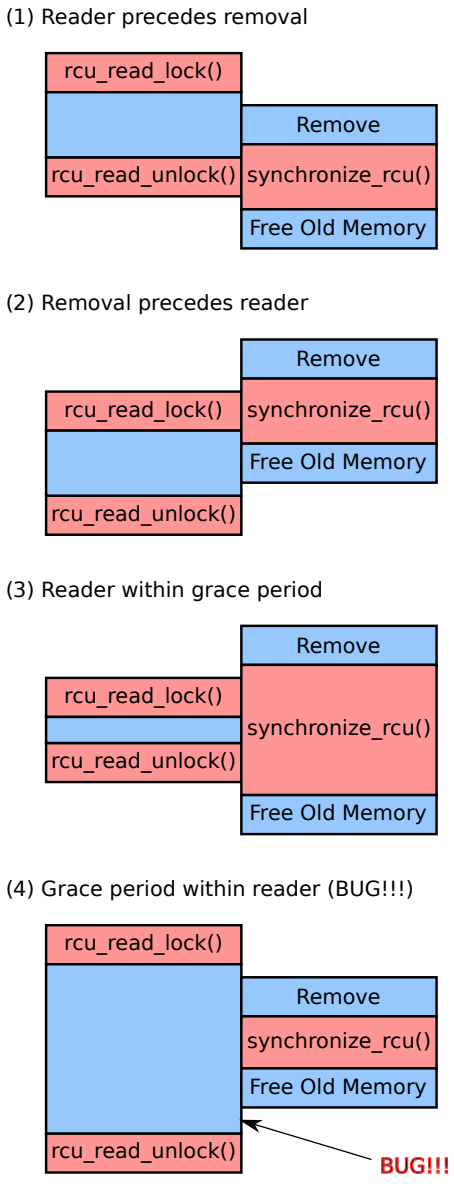


Figure 9.14: Summary of RCU Grace-Period Ordering Guarantees

the left-hand stack of boxes and and RCU updater by the right-hand stack.

In the first scenario, the reader starts execution before the updater starts the removal, so it is possible that this reader has a reference to the removed data element. Therefore, the updater must not free this element until after the reader completes. In the second scenario, the reader does not start execution until after the removal has completed. The reader cannot possibly obtain a reference to the already-removed data element, so this element may be freed before the reader completes. The third scenario is like the second, but illustrates that when the reader cannot possibly obtain a reference to element, it is still permissible to defer the freeing of that element until after the reader completes. In the fourth and final scenario, the reader starts execution before the updater starts removing the data element, but this element is (incorrectly) freed before the reader completed. A correct RCU implementation will not allow this fourth scenario to occur. This diagram thus illustrates RCU's wait-for-readers functionality: Given a grace period, each reader ends before the end of that grace period, starts after the beginning of that grace period, or both, in which case it is wholly contained within that grace period.

Because RCU readers can make forward progress while updates are in progress, different readers might disagree about the state of the data structure, a topic taken up by the next section.

9.5.2.3 Maintain Multiple Versions of Recently Updated Objects

This section discusses how RCU accommodates synchronization-free readers by maintaining multiple versions of data. This discussion builds on the introduction of multiple versions by Figure 9.7 in Section 9.5.1.1, in which readers running concurrently with `del_route()` (see Listing 9.13) might see the old `route` structure or an empty list, but either way get a valid result. Of course, a closer look at Figure 9.6 shows that calls to `ins_route()` can also result in concurrent readers seeing different versions: Either the initial empty list or the newly inserted `route` structure. Note that both reference counting (Section 9.2) and hazard pointers (Section 9.3) can also cause concurrent readers to see different versions, but RCU's lightweight readers make this more likely.

However, maintaining multiple versions can be even more surprising. For example, consider Figure 9.15, in which a reader is traversing a linked list that is concurrently

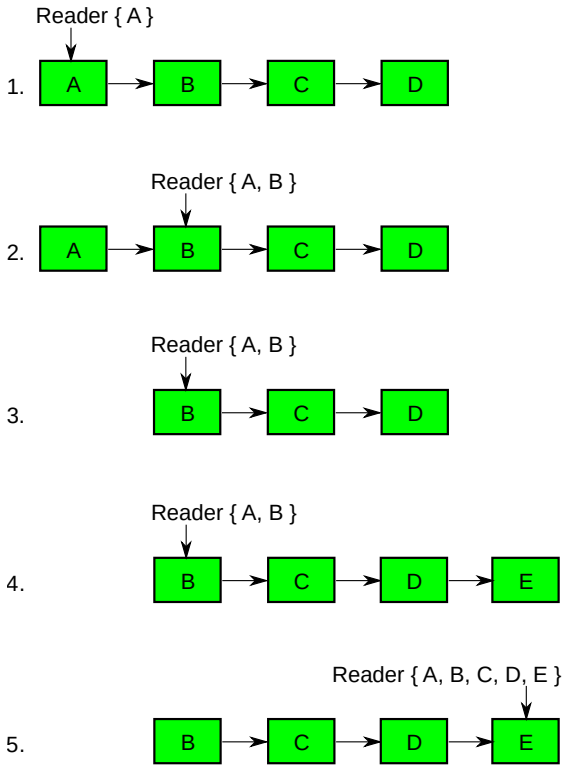


Figure 9.15: Multiple RCU Data-Structure Versions

updated.[11] In the first row of the figure, the reader is referencing data item A, and in the second row, it advances to B, having thus far seen A followed by B. In the third row, an updater removes element A and in the fourth row an updater adds element E to the end of the list. In the fifth and final row, the reader completes its traversal, having seeing elements A through E.

Except that there was no time at which such a list existed. This situation might be even more surprising than that shown in Figure 9.7, in which different concurrent readers see different versions. In contrast, in Figure 9.15 the reader sees a version that never actually existed!

One way to resolve this strange situation is via weaker semanitics. A reader traversal must encounter any data item that was present during the full traversal (B, C, and D), and might or might not encounter data items that were present for only part of the traversal (A and E). Therefore, in this particular case, it is perfectly legitimate for the reader traveral to encounter all five elements. If this outcome is problematic, another way to resolve this situation is through use of stronger synchronization mechanisms, such as reader-writer locking or clever use of timestamps or versioning. Of course, stronger mechanisms will be more expensive, but then again the engineering life is all about choices and tradeoffs.

Strange though this situation might seem, it is entirely consistent with the real world. As we saw in Section 3.2, the finite speed of light cannot be ignored within a computer system, and it most certainly cannot be ignored outside of this system. This in turn means that any data within the system representing state in the real world outside of the system is always and forever outdated, and thus inconsistent with the real world. As a result, algorithms operating on real-world data must account for inconsistent data. In many cases, these algorithms are also perfectly capable of dealing with inconsistencies within the system.

The pre-BSD packet routing example laid out in Section 9.1 is a case in point. The contents of a routing list is set by routing protocols, and these protocols feature significant delays (seconds or even minutes) to avoid routing instabilities. Therefore, once a routing update reaches a given system, it might well have been sending packets the wrong way for quite some time. Sending a few more packets the wrong way for the few microseconds during which the update is in flight is clearly not a problem because the same higher-level protocol actions that deal with delayed routing updates will also deal with internal inconsistencies.

Nor is Internet routing the only situation tolerating inconsistencies. To repeat, any algorithm in which data within a system tracks outside-of-system state must tolerate inconsistencies, which includes security policies (often set by committees of humans), storage configuration, and WiFi access points, to say nothing of removable hardware such as microphones, headsets, cameras, mice, printers, and much else besides. Furthermore, the large number of Linux-kernel RCU API uses shown in Figure 9.9, combined with the Linux kernel's heavy use of reference counting and with increasing use of hazard pointers in other projects, demonstrates that tolerance for such inconsistencies is more common than one might imagine. This is especially the case given that single-item lookups are much more common than traversals: After all, (1) concurrent updates are less likely to affect a single-item lookup than they are a full traversal, and (2) an isolated single-item lookup cannot detect such inconsistencies.

From a more theoretical viewpoint, there are even some special cases where RCU readers can be considered to be fully ordered with updaters, despite the fact that

[11] RCU linked-list APIs may be found in Section 9.5.3.

www.dbooks.org

these readers might be executing the exact same sequence of machine instructions that would be executed by a single-threaded program. For example, referring back to Listing 9.13 on page 139, suppose that each reader thread invokes `access_route()` exactly once during its lifetime, and that there is no other communication among reader and updater threads. Then each invocation of `access_route()` can be ordered after the `ins_route()` invocation that produced the `route` structure accessed by line 11 of the listing in `access_route()` and ordered before any subsequent `ins_route()` or `del_route()` invocation.

In summary, maintaining multiple versions is exactly what enables the extremely low overheads of RCU readers, and as noted earlier, many algorithms are unfazed by multiple versions. However, there are algorithms that absolutely cannot handle multiple versions. There are techniques for adapting such algorithms to RCU [McK04], but these are beyond the scope of this section.

Exercises These examples assumed that a mutex was held across the entire update operation, which would mean that there could be at most two versions of the list active at a given time.

Quick Quiz 9.33: How would you modify the deletion example to permit more than two versions of the list to be active? ■

Quick Quiz 9.34: How many RCU versions of a given list can be active at any given time? ■

9.5.2.4 Summary of RCU Fundamentals

This section has described the three fundamental components of RCU-based algorithms:

1. a publish-subscribe mechanism for adding new data,
2. a way of waiting for pre-existing RCU readers to finish (see Section 15.4.2 for more detail), and
3. a discipline of maintaining multiple versions to permit change without harming or unduly delaying concurrent RCU readers.

Quick Quiz 9.35: How can RCU updaters possibly delay RCU readers, given that neither `rcu_read_lock()` nor `rcu_read_unlock()` spin or block? ■

These three RCU components allow data to be updated in face of concurrent readers that might be executing the same sequence of machine instructions that would be used by a reader in a single-threaded implementation. These RCU components can be combined in different ways to implement a surprising variety of different types of RCU-based algorithms. However, it is usually better to work at higher levels of abstraction. To this end, the next section describes the Linux-kernel API, which includes simple data structures such as lists.

9.5.3 RCU Linux-Kernel API

This section looks at RCU from the viewpoint of its Linux-kernel API.[12] Section 9.5.3.1 presents RCU's wait-to-finish APIs, Section 9.5.3.2 presents RCU's publish-subscribe and version-maintenance APIs, Section 9.5.3.3 presents RCU's list-processing APIs, Section 9.5.3.4 presents RCU's diagnostic APIs, and Section 9.5.3.5 describes in which contexts RCU's various APIs may be used. Finally, Section 9.5.3.6 presents concluding remarks.

Readers who are not excited about kernel internals may wish to skip ahead to Section 9.5.4 on page 155.

9.5.3.1 RCU has a Family of Wait-to-Finish APIs

The most straightforward answer to "what is RCU" is that RCU is an API. For example, the RCU implementation used in the Linux kernel is summarized by Table 9.1, which shows the wait-for-readers portions of the RCU, "sleepable" RCU (SRCU), Tasks RCU, and generic APIs, respectively, and by Table 9.2, which shows the publish-subscribe portions of the API [McK19b].[13]

If you are new to RCU, you might consider focusing on just one of the columns in Table 9.1, each of which summarizes one member of the Linux kernel's RCU API family. For example, if you are primarily interested in understanding how RCU is used in the Linux kernel, "RCU" would be the place to start, as it is used most frequently. On the other hand, if you want to understand RCU for its own sake, "Task RCU" has the simplest API. You can always come back for the other columns later.

If you are already familiar with RCU, these tables can serve as a useful reference.

Quick Quiz 9.36: Why do some of the cells in Table 9.1 have exclamation marks ("!")? ■

[12] Userspace RCU's API is documented elsewhere [MDJ13c].

[13] This citation covers v4.20 and later. Documetation for earlier versions of the Linux-kernel RCU API may be found elsewhere [McK08d, McK14d].

Table 9.1: RCU Wait-to-Finish APIs

	RCU: Original	**SRCU**: Sleeping readers	**Tasks RCU**: Free tracing trampolines	**Tasks RCU Rude**: Free idle-task tracing trampolines	**Tasks RCU Trace**: Protect sleepable BPF programs
Read-side critical-section markers	`rcu_read_lock()` ! `rcu_read_unlock()` ! `rcu_read_lock_bh()` `rcu_read_unlock_bh()` `rcu_read_lock_sched()` `rcu_read_unlock_sched()` (Plus anything disabing bottom halves, preemption, or interrupts.)	`srcu_read_lock()` `srcu_read_unlock()`	Voluntary context switch	Voluntary context switch and preempt-enable regions of code	`rcu_read_lock_trace()` `rcu_read_unlock_trace()`
Update-side primitives (synchronous)	`synchronize_rcu()` `synchronize_net()` `synchronize_rcu_expedited()`	`synchronize_srcu()` `synchronize_srcu_expedited()`	`synchronize_rcu_tasks()`	`synchronize_rcu_tasks_rude()`	`synchronize_rcu_tasks_trace()`
Update-side primitives (asynchronous / callback)	`call_rcu()` !	`call_srcu()`	`call_rcu_tasks()`	`call_rcu_tasks_rude()`	`call_rcu_tasks_trace()`
Update-side primitives (wait for callbacks)	`rcu_barrier()`	`srcu_barrier()`	`rcu_barrier_tasks()`	`rcu_barrier_tasks_rude()`	`rcu_barrier_tasks_trace()`
Update-side primitives (initiate / wait)	`get_state_synchronize_rcu()` `cond_synchronize_rcu()`				
Update-side primitives (free memory)	`kfree_rcu()`				
Type-safe memory	`SLAB_TYPESAFE_BY_RCU`				
Read side constraints	No blocking (only preemption)	No `synchronize_srcu()` with same `srcu_struct`	No voluntary context switch	Neither blocking nor preemption	No RCU tasks trace grace period
Read side overhead	CPU-local accesses (`barrier()` on `PREEMPT=n`)	Simple instructions, memory barriers	Free	CPU-local accesses (free on `PREEMPT=n`)	CPU-local accesses
Asynchronous update-side overhead	sub-microsecond	sub-microsecond	sub-microsecond	sub-microsecond	sub-microsecond
Grace-period latency	10s of milliseconds	Milliseconds	Seconds	Milliseconds	10s of milliseconds
Expedited grace-period latency	10s of microseconds	Microseconds	N/A	N/A	N/A

www.dbooks.org

The "RCU" column corresponds to the consolidation of the three Linux-kernel RCU implementations [McK19c, McK19a], in which RCU read-side critical sections start with `rcu_read_lock()`, `rcu_read_lock_bh()`, or `rcu_read_lock_sched()` and end with `rcu_read_unlock()`, `rcu_read_unlock_bh()`, or `rcu_read_unlock_sched()`, respectively. Any region of code that disables bottom halves, interrupts, or preemption also acts as an RCU read-side critical section. RCU read-side critical sections may be nested. The corresponding synchronous update-side primitives, `synchronize_rcu()` and `synchronize_rcu_expedited()`, along with their synonym `synchronize_net()`, wait for any type of currently executing RCU read-side critical sections to complete. The length of this wait is known as a "grace period", and `synchronize_rcu_expedited()` is designed to reduce grace-period latency at the expense of increased CPU overhead and IPIs. The asynchronous update-side primitive, `call_rcu()`, invokes a specified function with a specified argument after a subsequent grace period. For example, `call_rcu(p,f);` will result in the "RCU callback" `f(p)` being invoked after a subsequent grace period. There are situations, such as when unloading a Linux-kernel module that uses `call_rcu()`, when it is necessary to wait for all outstanding RCU callbacks to complete [McK07e]. The `rcu_barrier()` primitive does this job.

Quick Quiz 9.37: How do you prevent a huge number of RCU read-side critical sections from indefinitely blocking a `synchronize_rcu()` invocation? ■

Quick Quiz 9.38: The `synchronize_rcu()` API waits for all pre-existing interrupt handlers to complete, right? ■

Finally, RCU may be used to provide type-safe memory [GC96], as described in Section 9.5.4.8. In the context of RCU, type-safe memory guarantees that a given data element will not change type during any RCU read-side critical section that accesses it. To make use of RCU-based type-safe memory, pass `SLAB_TYPESAFE_BY_RCU` to `kmem_cache_create()`.

The "SRCU" column in Table 9.1 displays a specialized RCU API that permits general sleeping in SRCU read-side critical sections [McK06] delimited by `srcu_read_lock()` and `srcu_read_unlock()`. However, unlike RCU, SRCU's `srcu_read_lock()` returns a value that must be passed into the corresponding `srcu_read_unlock()`. This difference is due to the fact that the SRCU user allocates an `srcu_struct` for each distinct SRCU usage. These distinct `srcu_struct` structures prevent SRCU read-side critical sections from blocking unrelated `synchronize_srcu()` and `synchronize_srcu_expedited()` invocations. Of course, use of either `synchronize_srcu()` or `synchronize_srcu_expedited()` within an SRCU read-side critical section can result in self-deadlock, so should be avoided. As with RCU, SRCU's `synchronize_srcu_expedited()` decreases grace-period latency compared to `synchronize_srcu()`, but at the expense of increased CPU overhead.

Quick Quiz 9.39: Under what conditions can `synchronize_srcu()` be safely used within an SRCU read-side critical section? ■

Similar to normal RCU, self-deadlock can be avoided using the asynchronous `call_srcu()` function. However, special care must be taken when using `call_srcu()` because a single task could register SRCU callbacks very quickly. Given that SRCU allows readers to block for arbitrary periods of time, this could consume an arbitrarily large quantity of memory. In contrast, given the synchronous `synchronize_srcu()` interface, a given task must finish waiting for a given grace period before it can start waiting for the next one.

Also similar to RCU, there is an `srcu_barrier()` function that waits for all prior `call_srcu()` callbacks to be invoked.

In other words, SRCU compensates for its extremely weak forward-progress guarantees by permitting the developer to restrict its scope.

The "Tasks RCU" column in Table 9.1 displays a specialized RCU API that mediates freeing of the trampolines used in Linux-kernel tracing. These trampolines are used to transfer control from a point in the code being traced to the code doing the actual tracing. It is of course necessary to ensure that all code executing within a given trampoline has finished before freeing that trampoline.

Changes to the code being traced are typically limited to a single jump or call instruction, and thus cannot accommodate the sequence of code required to implement `rcu_read_lock()` and `rcu_read_unlock()`. Nor can the trampoline contain calls to `rcu_read_lock()` and `rcu_read_unlock()`. To see this, consider a CPU that is just about to start executing a given trampoline. Because it has not yet executed the `rcu_read_lock()`, that trampoline could be freed at any time, which would come as a fatal surprise to this CPU. Therefore, trampolines cannot be protected by synchronization primitives executed in either the traced code or in the trampoline itself. Which does raise the question of exactly how the trampoline is to be protected.

The key to answering this question is to note that trampoline code never contains code that either directly or indirectly does a voluntary context switch. This code might be preempted, but it will never directly or indirectly invoke `schedule()`. This suggests a variant of RCU having voluntary context switches and idle execution as its only quiescent states. This variant is Tasks RCU.

Tasks RCU is unusual in having no read-side marking functions, which is good given that its main use case has nowhere to put such markings. Instead, calls to `schedule()` serve directly as quiescent states. Updates can use `synchronize_rcu_tasks()` to wait for all pre-existing trampoline execution to complete, or they can use its asynchronous counterpart, `call_rcu_tasks()`. There is also an `rcu_barrier_tasks()` that waits for completion of callbacks corresponding to all prior invocations of `call_rcu_tasks()`. There is no `synchronize_rcu_tasks_expedited()` because there has not yet been a request for it, though implementing a useful variant of it would not be free of challenges.

Quick Quiz 9.40: In a kernel built with `CONFIG_PREEMPT_NONE=y`, won't `synchronize_rcu()` wait for all trampolines, given that preemption is disabled and that trampolines never directly or indirectly invoke `schedule()`? ■

The "Tasks RCU Rude" column provides a more effective variant of the toy implementation presented in Section 9.5.1.3. This variant causes each CPU to execute a context switch, so that any voluntary context switch or any preemptible region of code can serve as a quiescent state. The Tasks RCU Rude variant uses the Linux-kernel workqueues facility to force concurrent context switches, in contrast to the serial CPU-by-CPU approach taken by the toy implementation. The API mirrors that of Tasks RCU, including the lack of explicit read-side markers.

Finally, the "Tasks RCU Trace" column provides an RCU implementation with functionality similar to that of SRCU, except with much faster read-side markers.[14] However, this speed is a consequence of the fact that these markers do not execute memory-barrier instructions, which means that Tasks RCU Trace grace periods must often send IPIs to all CPUs and must always scan the entire task list. Nevertheless, the resulting grace-period latency is reasonably short, rivaling that of RCU.

[14] And thus is unusual for the Tasks RCU family for having explicit read-side markers!

9.5.3.2 RCU has Publish-Subscribe and Version-Maintenance APIs

Fortunately, the RCU publish-subscribe and version-maintenance primitives shown in Table 9.2 apply to all of the variants of RCU discussed above. This commonality can allow more code to be shared, and reduces API proliferation. The original purpose of the RCU publish-subscribe APIs was to bury memory barriers into these APIs, so that Linux kernel programmers could use RCU without needing to become expert on the memory-ordering models of each of the 20+ CPU families that Linux supports [Spr01].

These primitives operate directly on pointers, and are useful for creating RCU-protected linked data structures, such as RCU-protected arrays and trees. The special case of linked lists is handled by a separate set of APIs described in Section 9.5.3.3.

The first category publishes pointers to new data items. The `rcu_assign_pointer()` primitive ensures that any prior initialization remains ordered before the assignment to the pointer on weakly ordered machines. The `rcu_replace_pointer()` primitive updates the pointer just like `rcu_assign_pointer()` does, but also returns the previous value, just like `rcu_dereference_protected()` (see below) would, including the lockdep expression. This replacement is convenient when the updater must both publish a new pointer and free the structure referenced by the old pointer.

Quick Quiz 9.41: Normally, any pointer subject to `rcu_dereference()` *must* always be updated using one of the pointer-publish functions in Table 9.2, for example, `rcu_assign_pointer()`.
What is an exception to this rule? ■

Quick Quiz 9.42: Are there any downsides to the fact that these traversal and update primitives can be used with any of the RCU API family members? ■

The `rcu_pointer_handoff()` primitive simply returns its sole argument, but is useful to tooling checking for pointers being leaked from RCU read-side critical sections. Use of `rcu_pointer_handoff()` indicates to such tooling that protection of the structure in question has been handed off from RCU to some other mechanism, such as locking or reference counting.

The `RCU_INIT_POINTER()` macro can be used to initialize RCU-protected pointers that have not yet been exposed to readers, or alternatively, to set RCU-protected pointers to `NULL`. In these restricted cases, the memory-barrier instructions provided by `rcu_assign_`

www.dbooks.org

Table 9.2: RCU Publish-Subscribe and Version Maintenance APIs

Category	Primitives	Overhead
Pointer publish	`rcu_assign_pointer()`	Memory barrier
	`rcu_replace_pointer()`	Memory barrier (two of them on Alpha)
	`rcu_pointer_handoff()`	Simple instructions
	`RCU_INIT_POINTER()`	Simple instructions
	`RCU_POINTER_INITIALIZER()`	Compile-time constant
Pointer subscribe (traveral)	`rcu_access_pointer()`	Simple instructions
	`rcu_dereference()`	Simple instructions (memory barrier on Alpha)
	`rcu_dereference_check()`	Simple instructions (memory barrier on Alpha)
	`rcu_dereference_protected()`	Simple instructions
	`rcu_dereference_raw()`	Simple instructions (memory barrier on Alpha)
	`rcu_dereference_raw_notrace()`	Simple instructions (memory barrier on Alpha)

`pointer()` are not needed. Similarly, `RCU_POINTER_INITIALIZER()` provides a GCC-style structure initializer to allow easy initialization of RCU-protected pointers in structures.

The second category subscribes to pointers to data items, or, alternatively, safely traverses RCU-protected pointers. Again, simply loading these pointers using C-language accesses could result in seeing pre-initialization garbage in the pointed-to data. Similarly, loading these pointer by any means outside of an RCU read-side critical section could result in the pointed-to object being freed at any time. However, if the pointer is merely to be tested and not dereferenced, the freeing of the pointed-to object is not necessarily a problem. In this case, `rcu_access_pointer()` may be used. Normally, however, RCU read-side protection is required, and so the `rcu_dereference()` primitive uses the Linux kernel's `lockdep` facility [Cor06a] to verify that this `rcu_dereference()` invocation is under the protection of `rcu_read_lock()`, `srcu_read_lock()`, or some other RCU read-side marker. In contrast, the `rcu_access_pointer()` primitive does not involve `lockdep`, and thus will not provoke `lockdep` complaints when used outside of an RCU read-side critical section.

Another situation where protection is not required is when update-side code accesses the RCU-protected pointer while holding the update-side lock. The `rcu_dereference_protected()` API member is provided for this situation. Its first parameter is the RCU-protected pointer, and the second parameter takes a lockdep expression describing which locks must be held in order for the access to be safe. Code invoked both from readers and updaters can use `rcu_dereference_check()`, which also takes a lockdep expression, but which may also be invoked from read-side code not holding the locks. In some cases, the lockdep expressions can be very complex, for example, when using fine-grained locking, any of a very large number of locks might be held, and it might be quite difficult to work out which applies. In these (hopefully rare) cases, `rcu_dereference_raw()` provides protection but does not check for being invoked within a reader or with any particular lock being held. The `rcu_dereference_raw_notrace()` API member acts similarly, but cannot be traced, and may therefore be safely used by tracing code.

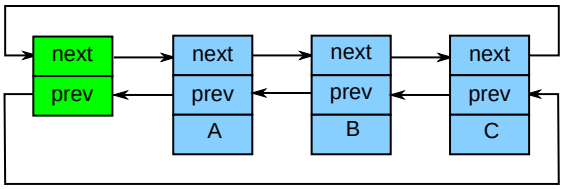


Figure 9.16: Linux Circular Linked List (`list`)

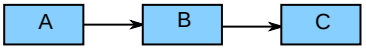


Figure 9.17: Linux Linked List Abbreviated

Although pretty much any linked structure can be accessed by manipulating pointers, higher-level structures can be quite helpful. The next section therefore looks at various sorts of RCU-protected linked lists used by the Linux kernel.

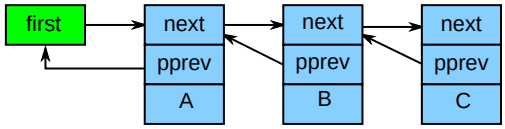


Figure 9.18: Linux Linear Linked List (`hlist`)

9.5.3.3 RCU has List-Processing APIs

Although `rcu_assign_pointer()` and `rcu_dereference()` can in theory be used to construct any conceivable RCU-protected data structure, in practice it is often better to use higher-level constructs. Therefore, the `rcu_assign_pointer()` and `rcu_dereference()` primitives have been embedded in special RCU variants of Linux's list-manipulation API. Linux has four variants of doubly linked list, the circular `struct list_head` and the linear `struct hlist_head/struct hlist_node`, `struct hlist_nulls_head/struct hlist_nulls_node`, and `struct hlist_bl_head/struct hlist_bl_node` pairs. The former is laid out as shown in Figure 9.16, where the green (leftmost) boxes represent the list header and the blue (rightmost three) boxes represent the elements in the list. This notation is cumbersome, and will therefore be abbreviated as shown in Figure 9.17, which shows only the non-header (blue) elements.

Linux's `hlist`[15] is a linear list, which means that it needs only one pointer for the header rather than the two required for the circular list, as shown in Figure 9.18. Thus, use of `hlist` can halve the memory consumption for the hash-bucket arrays of large hash tables. As before, this notation is cumbersome, so `hlist` structures will be abbreviated in the same way `list_head`-style lists are, as shown in Figure 9.17.

A variant of Linux's `hlist`, named `hlist_nulls`, provides multiple distinct `NULL` pointers, but otherwise uses the same layout as shown in Figure 9.18. In this variant, a `->next` pointer having a zero low-order bit is considered to be a pointer. However, if the low-order bit is set to one, the upper bits identify the type of `NULL` pointer. This type of list is used to allow lockless readers to detect when a node has been moved from one list to another. For example, each bucket of a hash table might use its index to mark its `NULL` pointer. Should a reader encounter a `NULL` pointer not matching the index of the bucket it started from, that reader knows that an element it was traversing was moved to some other bucket during the traversal, taking that reader with it. The reader can use the `is_a_nulls()` function (which returns true if passed an `hlist_nulls` `NULL` pointer) to determine when it reaches the end of a list, and the `get_nulls_value()` function (which returns its argument's `NULL`-pointer identifier) to fetch the type of `NULL` pointer. When `get_nulls_values()` returns an unexpected value, the reader can take corrective action, for example, restarting its traversal from the beginning.

Quick Quiz 9.43: But what if an `hlist_nulls` reader gets moved to some other bucket and then back again? ■

More information on `hlist_nulls` is available in the Linux-kernel source tree, with helpful example code provided in the `rculist_nulls.rst` file (`rculist_nulls.txt` in older kernels).

Another variant of Linux's `hlist` incorporates bit-locking, and is named `hlist_bl`. This variant uses the same layout as shown in Figure 9.18, but reserves the low-order bit of the head pointer ("first" in the figure) to lock the list. This approach also reduces memory usage, as it allows what would otherwise be a separate spinlock to be stored with the pointer itself.

The API members for these linked-list variants are summarized in Table 9.3. More information is available in the `Documentation/RCU` directory of the Linux-kernel source tree and at Linux Weekly News [McK19b].

However, the remainder of this section expands on the use of `list_replace_rcu()`, given that this API member gave RCU its name. This API member is used to carry out more complex updates in which an element in the middle of the list having multiple fields is atomically updated, so that a given reader sees either the old set of values or the new set of values, but not a mixture of the two sets. For example, each node of a linked list might have integer fields `->a`, `->b`, and `->c`, and it might be necessary to update a given node's fields from 5, 6, and 7 to 5, 2, and 3, respectively.

The code implementing this atomic update is straightforward:

```
15 q = kmalloc(sizeof(*p), GFP_KERNEL);
16 *q = *p;
17 q->b = 2;
18 q->c = 3;
19 list_replace_rcu(&p->list, &q->list);
20 synchronize_rcu();
21 kfree(p);
```

The following discussion walks through this code, using Figure 9.19 to illustrate the state changes. The triples in each element represent the values of fields `->a`, `->b`,

[15] The "h" stands for hashtable, in which it reduces memory use by half compared to Linux's double-pointer circular linked list.

www.dbooks.org

Table 9.3: RCU-Protected List APIs

list: Circular doubly linked list	**hlist**: Linear doubly linked list	**hlist_nulls**: Linear doubly linked list with marked NULL pointer, with up to 31 bits of marking	**hlist_bl**: Linear doubly linked list with bit locking
Structures			
`struct list_head`	`struct hlist_head` `struct hlist_node`	`struct hlist_nulls_head` `struct hlist_nulls_node`	`struct hlist_bl_head` `struct hlist_bl_node`
Initialization			
	`INIT_LIST_HEAD_RCU()`		
Full traversal			

`list_for_each_entry_rcu()`
`list_for_each_entry_lockless()`
	`hlist_for_each_entry_rcu()`
	`hlist_for_each_entry_rcu_bh()`
	`hlist_for_each_entry_rcu_notrace()`
		`hlist_nulls_for_each_entry_rcu()`
		`hlist_nulls_for_each_entry_safe()`
			`hlist_bl_for_each_entry_rcu()`

Resume traversal			

`list_for_each_entry_continue_rcu()`
`list_for_each_entry_from_rcu()`
	`hlist_for_each_entry_continue_rcu()`
	`hlist_for_each_entry_continue_rcu_bh()`
	`hlist_for_each_entry_from_rcu()`		

Stepwise traversal			

`list_entry_rcu()`
`list_entry_lockless()`
`list_first_or_null_rcu()`
`list_next_rcu()`
`list_next_or_null_rcu()`
	`hlist_first_rcu()`
	`hlist_next_rcu()`
	`hlist_pprev_rcu()`
		`hlist_nulls_first_rcu()`
		`hlist_nulls_next_rcu()`
			`hlist_bl_first_rcu()`

Add			

`list_add_rcu()`
`list_add_tail_rcu()`
	`hlist_add_before_rcu()`
	`hlist_add_behind_rcu()`
	`hlist_add_head_rcu()`
	`hlist_add_tail_rcu()`
		`hlist_nulls_add_head_rcu()`	`hlist_bl_add_head_rcu()`
			`hlist_bl_set_first_rcu()`

Delete			
`list_del_rcu()`	`hlist_del_rcu()` `hlist_del_init_rcu()`	`hlist_nulls_del_rcu()` `hlist_nulls_del_init_rcu()`	`hlist_bl_del_rcu()` `hlist_bl_del_init_rcu()`
Replace			
`list_replace_rcu()`	`hlist_replace_rcu()`		
Splice			
`list_splice_init_rcu()`	`list_splice_tail_init_rcu()`		

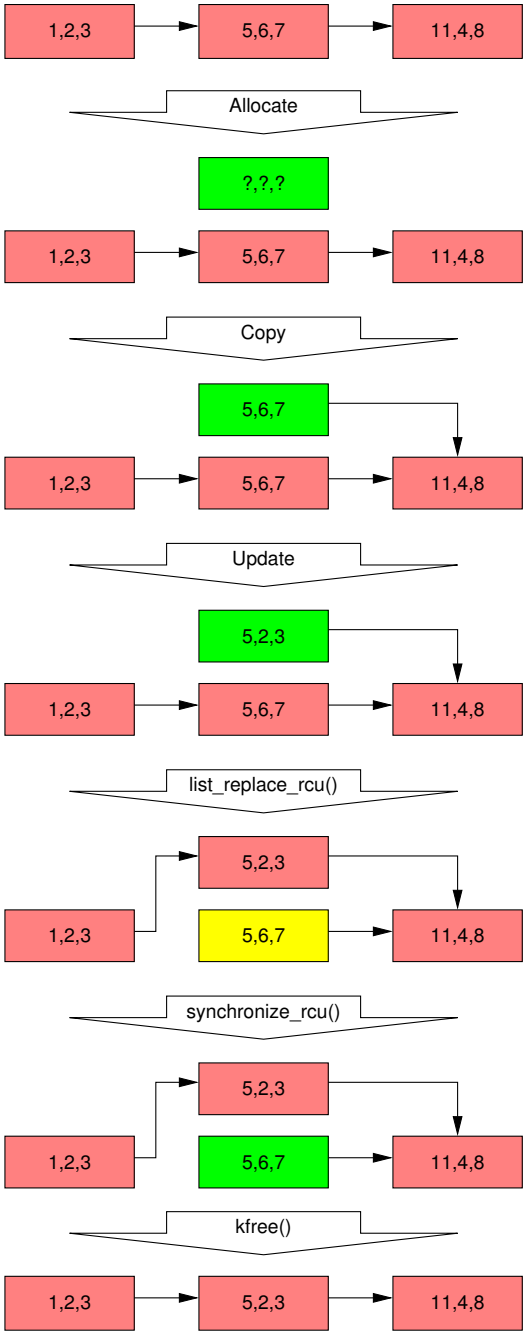


Figure 9.19: RCU Replacement in Linked List

and `->c`, respectively. The red-shaded elements might be referenced by readers, and because readers do not synchronize directly with updaters, readers might run concurrently with this entire replacement process. Please note that backwards pointers and the link from the tail to the head are omitted for clarity.

The initial state of the list, including the pointer `p`, is the same as for the deletion example, as shown on the first row of the figure.

The following text describes how to replace the `5,6,7` element with `5,2,3` in such a way that any given reader sees one of these two values.

Line 15 allocates a replacement element, resulting in the state as shown in the second row of Figure 9.19. At this point, no reader can hold a reference to the newly allocated element (as indicated by its green shading), and it is uninitialized (as indicated by the question marks).

Line 16 copies the old element to the new one, resulting in the state as shown in the third row of Figure 9.19. The newly allocated element still cannot be referenced by readers, but it is now initialized.

Line 17 updates `q->b` to the value "2", and line 18 updates `q->c` to the value "3", as shown on the fourth row of Figure 9.19. Note that the newly allocated structure is still inaccessible to readers.

Now, line 19 does the replacement, so that the new element is finally visible to readers, and hence is shaded red, as shown on the fifth row of Figure 9.19. At this point, as shown below, we have two versions of the list. Pre-existing readers might see the `5,6,7` element (which is therefore now shaded yellow), but new readers will instead see the `5,2,3` element. But any given reader is guaranteed to see one set of values or the other, not a mixture of the two.

After the `synchronize_rcu()` on line 20 returns, a grace period will have elapsed, and so all reads that started before the `list_replace_rcu()` will have completed. In particular, any readers that might have been holding references to the `5,6,7` element are guaranteed to have exited their RCU read-side critical sections, and are thus prohibited from continuing to hold a reference. Therefore, there can no longer be any readers holding references to the old element, as indicated its green shading in the sixth row of Figure 9.19. As far as the readers are concerned, we are back to having a single version of the list, but with the new element in place of the old.

After the `kfree()` on line 21 completes, the list will appear as shown on the final row of Figure 9.19.

www.dbooks.org

Table 9.4: RCU Diagnostic APIs

Category	Primitives
Mark RCU pointer	`__rcu`

Debug-object support	`init_rcu_head()`
	`destroy_rcu_head()`
	`init_rcu_head_on_stack()`
	`destroy_rcu_head_on_stack()`

Stall-warning control	`rcu_cpu_stall_reset()`
Callback checking	`rcu_head_init()` `rcu_head_after_call_rcu()`

lockdep support	`rcu_read_lock_held()`
	`rcu_read_lock_bh_held()`
	`rcu_read_lock_sched_held()`
	`srcu_read_lock_held()`
	`rcu_is_watching()`
	`RCU_LOCKDEP_WARN()`
	`RCU_NONIDLE()`
	`rcu_sleep_check()`

Despite the fact that RCU was named after the replacement case, the vast majority of RCU usage within the Linux kernel relies on the simple independent insertion and deletion, as was shown in Figure 9.15 in Section 9.5.2.3.

The next section looks at APIs that assist developers in debugging their code that makes use of RCU.

9.5.3.4 RCU Has Diagnostic APIs

Table 9.4 shows RCU's diagnostic APIs.

The `__rcu` marks an RCU-protected pointer, for example, "`struct foo __rcu *p;`". Pointers that might be passed to `rcu_dereference()` can be marked, but pointers holding values returned from `rcu_dereference()` should not be. Providing these markings on variables, structure fields, function parameters, and return values allow the Linux kernel's `sparse` tool to detect situtations where RCU-protected pointers are incorrectly accessed using plain C-language loads and stores.

Debug-object support is automatic for any `rcu_head` structures that are part of a structure obtained from the Linux kernel's memory allocators, but those building their own special-purpose memory allocators can use `init_rcu_head()` and `destroy_rcu_head()` at allocation and free time, respectively. Those using `rcu_head` structures allocated on the function-call stack (it happens!) may use `init_rcu_head_on_stack()` before first use and `destroy_rcu_head_on_stack()` after last use, but before returning from the function. Debug-object support allows detection of bugs involving passing the same `rcu_head` structure to `call_rcu()` and friends in quick succession, which is the `call_rcu()` counterpart to the infamous double-free class of memory-allocation bugs.

Stall-warning control is provided by `rcu_cpu_stall_reset()`, which allows the caller to suppress RCU CPU stall warnings for the remainder of the current grace period. RCU CPU stall warnings help pinpoint situations where an RCU read-side critical section runs for an excessive length of time, and it is useful for things like kernel debuggers to be able to suppress them, for example, when encountering a breakpoint.

Callback checking is provided by `rcu_head_init()` and `rcu_head_after_call_rcu()`. The former is invoked on an `rcu_head` structure before it is passed to `call_rcu()`, and then `rcu_head_after_call_rcu()` will check to see if the callback is has been invoked with the specified function.

Support for lockdep [Cor06a] includes `rcu_read_lock_held()`, `rcu_read_lock_bh_held()`, `rcu_read_lock_sched_held()`, and `srcu_read_lock_held()`, each of which returns `true` if invoked within the corresponding type of RCU read-side critical section.

Quick Quiz 9.44: Why isn't there a `rcu_read_lock_tasks_held()` for Tasks RCU? ■

Because `rcu_read_lock()` cannot be used from the idle loop, and because energy-efficiency concerns have caused the idle loop to become quite ornate, `rcu_is_watching()` returns true if invoked in a context where use of `rcu_read_lock()` is legal. Note again that `srcu_read_lock()` may be used from idle and even offline CPUs, which means that `rcu_is_watching()` does not apply to SRCU.

`RCU_LOCKDEP_WARN()` emits a warning if lockdep is enabled and if its argument evaluated to `true`. For example, `RCU_LOCKDEP_WARN(!rcu_read_lock_held())` would emit a warning if invoked outside of an RCU read-side critical section.

`RCU_NONIDLE()` may be used to force RCU to watch when executing the statement that is passed in as the sole argument. For example, `RCU_NONIDLE(WARN_ON(!rcu_is_watching()))` would never emit a warning. However, changes in the 2020–2021 timeframe extend RCU's reach deeper into the idle loop, which should greatly reduce or even eliminate the need for `RCU_NONIDLE()`.

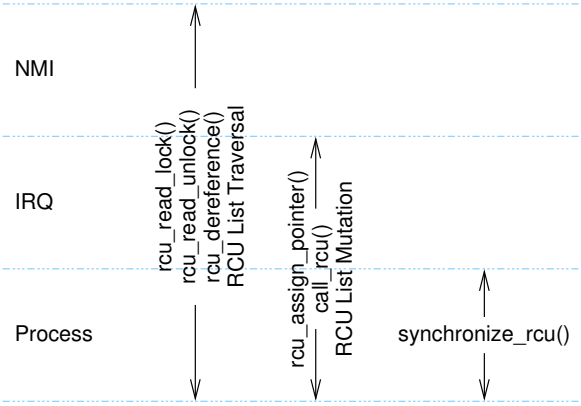


Figure 9.20: RCU API Usage Constraints

Finally, `rcu_sleep_check()` emits a warning if invoked within an RCU, RCU-bh, or RCU-sched read-side critical section.

9.5.3.5 Where Can RCU's APIs Be Used?

Figure 9.20 shows which APIs may be used in which in-kernel environments. The RCU read-side primitives may be used in any environment, including NMI, the RCU mutation and asynchronous grace-period primitives may be used in any environment other than NMI, and, finally, the RCU synchronous grace-period primitives may be used only in process context. The RCU list-traversal primitives include `list_for_each_entry_rcu()`, `hlist_for_each_entry_rcu()`, etc. Similarly, the RCU list-mutation primitives include `list_add_rcu()`, `hlist_del_rcu()`, etc.

Note that primitives from other families of RCU may be substituted, for example, `srcu_read_lock()` may be used in any context in which `rcu_read_lock()` may be used.

9.5.3.6 So, What *is* RCU Really?

At its core, RCU is nothing more nor less than an API that supports publication and subscription for insertions, waiting for all RCU readers to complete, and maintenance of multiple versions. That said, it is possible to build higher-level constructs on top of RCU, including the reader-writer-locking, reference-counting, and existence-guarantee constructs listed in Section 9.5.4. Furthermore, I have no doubt that the Linux community will continue to find interesting new uses for RCU, just as they do for any of a number of synchronization primitives throughout the kernel.

Table 9.5: RCU Usage

Mechanism RCU Replaces	Section
Reader-writer locking	Section 9.5.4.2
Restricted reference-counting	Section 9.5.4.3
Bulk reference-counting	Section 9.5.4.4
Garbage collector	Section 9.5.4.5
Multi-version concurrency control	Section 9.5.4.6
Existence Guarantees	Section 9.5.4.7
Type-Safe Memory	Section 9.5.4.8
Wait for things to finish	Section 9.5.4.9

Of course, a more-complete view of RCU would also include all of the things you can do with these APIs.

However, for many people, a complete view of RCU must include sample RCU implementations. Appendix B therefore presents a series of "toy" RCU implementations of increasing complexity and capability, though others might prefer the classic "User-Level Implementations of Read-Copy Update" [DMS+12]. For everyone else, the next section gives an overview of some RCU use cases.

9.5.4 RCU Usage

This section answers the question "What is RCU?" from the viewpoint of the uses to which RCU can be put. Because RCU is most frequently used to replace some existing mechanism, we look at it primarily in terms of its relationship to such mechanisms, as listed in Table 9.5. Following the sections listed in this table, Section 9.5.4.10 provides a summary.

9.5.4.1 RCU for Pre-BSD Routing

Listings 9.14 and 9.15 show code for an RCU-protected Pre-BSD routing table (`route_rcu.c`). The former shows data structures and `route_lookup()`, and the latter shows `route_add()` and `route_del()`.

In Listing 9.14, line 2 adds the `->rh` field used by RCU reclamation, line 6 adds the `->re_freed` use-after-free-check field, lines 16, 22, and 26 add RCU read-side protection, and lines 20 and 21 add the use-after-free check. In Listing 9.15, lines 11, 13, 30, 34, and 39 add update-side locking, lines 12 and 33 add RCU update-side protection, line 35 causes `route_cb()` to be invoked after a grace period elapses, and lines 17–24 define `route_cb()`. This

www.dbooks.org

Listing 9.14: RCU Pre-BSD Routing Table Lookup

```
1  struct route_entry {
2    struct rcu_head rh;
3    struct cds_list_head re_next;
4    unsigned long addr;
5    unsigned long iface;
6    int re_freed;
7  };
8  CDS_LIST_HEAD(route_list);
9  DEFINE_SPINLOCK(routelock);
10
11 unsigned long route_lookup(unsigned long addr)
12 {
13   struct route_entry *rep;
14   unsigned long ret;
15
16   rcu_read_lock();
17   cds_list_for_each_entry_rcu(rep, &route_list, re_next) {
18     if (rep->addr == addr) {
19       ret = rep->iface;
20       if (READ_ONCE(rep->re_freed))
21         abort();
22       rcu_read_unlock();
23       return ret;
24     }
25   }
26   rcu_read_unlock();
27   return ULONG_MAX;
28 }
```

Listing 9.15: RCU Pre-BSD Routing Table Add/Delete

```
1  int route_add(unsigned long addr, unsigned long interface)
2  {
3    struct route_entry *rep;
4
5    rep = malloc(sizeof(*rep));
6    if (!rep)
7      return -ENOMEM;
8    rep->addr = addr;
9    rep->iface = interface;
10   rep->re_freed = 0;
11   spin_lock(&routelock);
12   cds_list_add_rcu(&rep->re_next, &route_list);
13   spin_unlock(&routelock);
14   return 0;
15 }
16
17 static void route_cb(struct rcu_head *rhp)
18 {
19   struct route_entry *rep;
20
21   rep = container_of(rhp, struct route_entry, rh);
22   WRITE_ONCE(rep->re_freed, 1);
23   free(rep);
24 }
25
26 int route_del(unsigned long addr)
27 {
28   struct route_entry *rep;
29
30   spin_lock(&routelock);
31   cds_list_for_each_entry(rep, &route_list, re_next) {
32     if (rep->addr == addr) {
33       cds_list_del_rcu(&rep->re_next);
34       spin_unlock(&routelock);
35       call_rcu(&rep->rh, route_cb);
36       return 0;
37     }
38   }
39   spin_unlock(&routelock);
40   return -ENOENT;
41 }
```

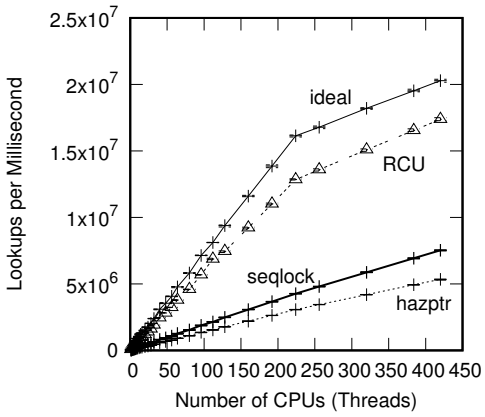


Figure 9.21: Pre-BSD Routing Table Protected by RCU

is minimal added code for a working concurrent implementation.

Figure 9.21 shows the performance on the read-only workload. RCU scales quite well, and offers nearly ideal performance. However, this data was generated using the `RCU_SIGNAL` flavor of userspace RCU [Des09b, MDJ13c], for which `rcu_read_lock()` and `rcu_read_unlock()` generate a small amount of code. What happens for the QSBR flavor of RCU, which generates no code at all for `rcu_read_lock()` and `rcu_read_unlock()`? (See Section 9.5.1, and especially Figure 9.8, for a discussion of RCU QSBR.)

The answer to this is shown in Figure 9.22, which shows that RCU QSBR's performance and scalability actually exceeds that of the ideal synchronization-free workload.

Quick Quiz 9.45: Wait, what??? How can RCU QSBR possibly be better than ideal? Just what rubbish definition of ideal would fail to be the best of all possible results??? ■

Quick Quiz 9.46: Given RCU QSBR's read-side performance, why bother with any other flavor of userspace RCU? ■

9.5.4.2 RCU is a Reader-Writer Lock Replacement

Perhaps the most common use of RCU within the Linux kernel is as a replacement for reader-writer locking in read-intensive situations. Nevertheless, this use of RCU was not immediately apparent to me at the outset, in fact, I chose to implement a lightweight reader-writer lock [HW92][16] before implementing a general-purpose

[16] Similar to `brlock` in the 2.4 Linux kernel and to `lglock` in more recent Linux kernels.

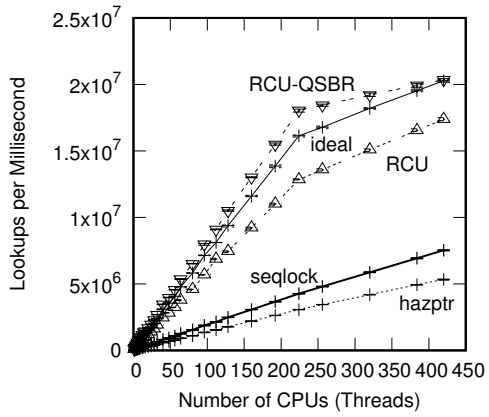


Figure 9.22: Pre-BSD Routing Table Protected by RCU QSBR

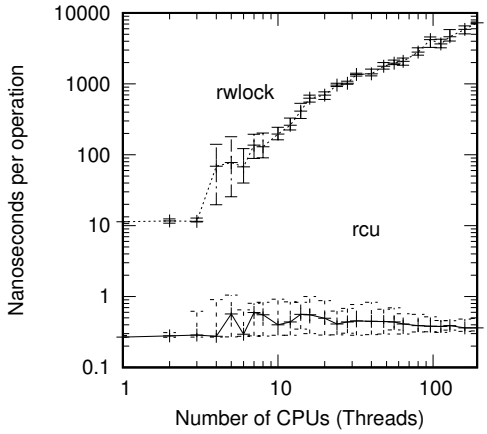


Figure 9.23: Performance Advantage of RCU Over Reader-Writer Locking

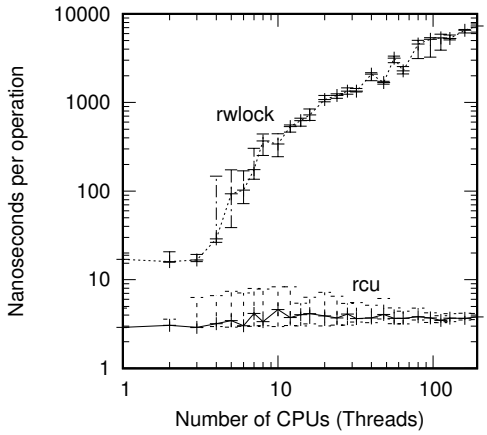


Figure 9.24: Performance Advantage of Preemptible RCU Over Reader-Writer Locking

RCU implementation back in the early 1990s. Each and every one of the uses I envisioned for the lightweight reader-writer lock was instead implemented using RCU. In fact, it was more than three years before the lightweight reader-writer lock saw its first use. Boy, did I feel foolish!

The key similarity between RCU and reader-writer locking is that both have read-side critical sections that can execute in parallel. In fact, in some cases, it is possible to mechanically substitute RCU API members for the corresponding reader-writer lock API members. But first, why bother?

Advantages of RCU include performance, deadlock immunity, and realtime latency. There are, of course, limitations to RCU, including the fact that readers and updaters run concurrently, that low-priority RCU readers can block high-priority threads waiting for a grace period to elapse, and that grace-period latencies can extend for many milliseconds. These advantages and limitations are discussed in the following sections.

Performance The read-side performance advantages of Linux-kernel RCU over reader-writer locking are shown in Figure 9.23, which was generated on a 448-CPU 2.10 GHz Intel x86 system.

Quick Quiz 9.47: WTF? How the heck do you expect me to believe that RCU can have less than a 300-picosecond overhead when the clock period at 2.10 GHz is almost 500 picoseconds? ■

Quick Quiz 9.48: Didn't an earlier release of this book show RCU read-side overhead way down in the sub-picosecond range? What happened??? ■

Quick Quiz 9.49: Why is there such large variation for the `rcu` trace in Figure 9.23? ■

Note that reader-writer locking is more than an order of magnitude slower than RCU on a single CPU, and is more than *four* orders of magnitude slower on 192 CPUs. In contrast, RCU scales quite well. In both cases, the error bars cover the full range of the measurements from 30 runs, with the line being the median.

A more moderate view may be obtained from a `CONFIG_PREEMPT` kernel, though RCU still beats reader-writer locking by between a factor of seven on a single CPU and by three orders of magnitude on 192 CPUs, as shown in Figure 9.24, which was generated on the same 448-CPU 2.10 GHz x86 system. Note the high variability of reader-

www.dbooks.org

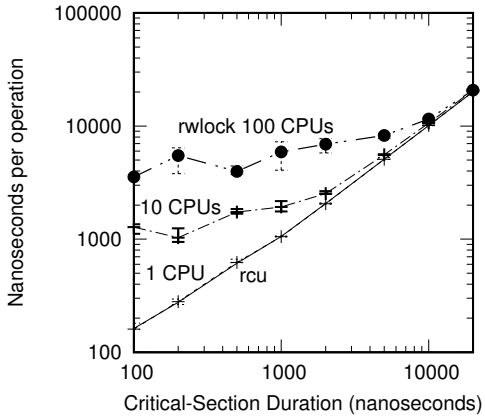


Figure 9.25: Comparison of RCU to Reader-Writer Locking as Function of Critical-Section Duration, 192 CPUs

writer locking at larger numbers of CPUs. The error bars span the full range of data.

Quick Quiz 9.50: Given that the system had no fewer than 448 hardware threads, why only 192 CPUs? ■

Of course, the low performance of reader-writer locking in Figures 9.23 and 9.24 is exaggerated by the unrealistic zero-length critical sections. The performance advantages of RCU decrease as the overhead of the critical sections increase. This decrease can be seen in Figure 9.25, which was run on the same system as the previous plots. Here, the y-axis represents the sum of the overhead of the read-side primitives and that of the critical section and the x-axis represents the critical-section overhead in nanoseconds. But please note the logscale y axis, which means that the small separations between the traces still represent significant differences. This figure shows non-preemptible RCU, but given that preemptible RCU's read-side overhead is only about three nanoseconds, its plot would be nearly identical to Figure 9.25.

Quick Quiz 9.51: Why the larger error ranges for the submicrosecond durations in Figure 9.25? ■

There are three traces for reader-writer locking, with the upper trace being for 100 CPUs, the next for 10 CPUs, and the lowest for 1 CPU. So the greater the number of CPUs and the shorter the critical sections, the greater is RCU's performance advantage. These performance advantages are underscored by the fact that 100-CPU systems are no longer uncommon and that a number of system calls (and thus any RCU read-side critical sections that they contain) complete within a microsecond.

In addition, as is discussed in the next section, RCU read-side primitives are almost entirely deadlock-immune.

Deadlock Immunity Although RCU offers significant performance advantages for read-mostly workloads, one of the primary reasons for creating RCU in the first place was in fact its immunity to read-side deadlocks. This immunity stems from the fact that RCU read-side primitives do not block, spin, or even do backwards branches, so that their execution time is deterministic. It is therefore impossible for them to participate in a deadlock cycle.

Quick Quiz 9.52: Is there an exception to this deadlock immunity, and if so, what sequence of events could lead to deadlock? ■

An interesting consequence of RCU's read-side deadlock immunity is that it is possible to unconditionally upgrade an RCU reader to an RCU updater. Attempting to do such an upgrade with reader-writer locking results in deadlock. A sample code fragment that does an RCU read-to-update upgrade follows:

```
1  rcu_read_lock();
2  list_for_each_entry_rcu(p, &head, list_field) {
3    do_something_with(p);
4    if (need_update(p)) {
5      spin_lock(my_lock);
6      do_update(p);
7      spin_unlock(&my_lock);
8    }
9  }
10 rcu_read_unlock();
```

Note that `do_update()` is executed under the protection of the lock *and* under RCU read-side protection.

Another interesting consequence of RCU's deadlock immunity is its immunity to a large class of priority inversion problems. For example, low-priority RCU readers cannot prevent a high-priority RCU updater from acquiring the update-side lock. Similarly, a low-priority RCU updater cannot prevent high-priority RCU readers from entering an RCU read-side critical section.

Quick Quiz 9.53: Immunity to both deadlock and priority inversion??? Sounds too good to be true. Why should I believe that this is even possible? ■

Realtime Latency Because RCU read-side primitives neither spin nor block, they offer excellent realtime latencies. In addition, as noted earlier, this means that they are

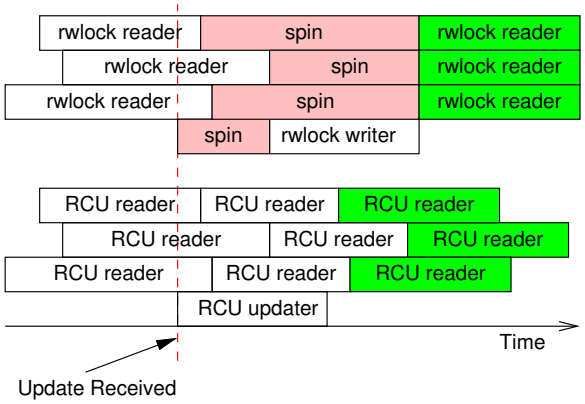


Figure 9.26: Response Time of RCU vs. Reader-Writer Locking

immune to priority inversion involving the RCU read-side primitives and locks.

However, RCU is susceptible to more subtle priority-inversion scenarios, for example, a high-priority process blocked waiting for an RCU grace period to elapse can be blocked by low-priority RCU readers in -rt kernels. This can be solved by using RCU priority boosting [McK07d, GMTW08].

RCU Readers and Updaters Run Concurrently Because RCU readers never spin nor block, and because updaters are not subject to any sort of rollback or abort semantics, RCU readers and updaters must necessarily run concurrently. This means that RCU readers might access stale data, and might even see inconsistencies, either of which can render conversion from reader-writer locking to RCU non-trivial.

However, in a surprisingly large number of situations, inconsistencies and stale data are not problems. The classic example is the networking routing table. Because routing updates can take considerable time to reach a given system (seconds or even minutes), the system will have been sending packets the wrong way for quite some time when the update arrives. It is usually not a problem to continue sending updates the wrong way for a few additional milliseconds. Furthermore, because RCU updaters can make changes without waiting for RCU readers to finish, the RCU readers might well see the change more quickly than would batch-fair reader-writer-locking readers, as shown in Figure 9.26.

Once the update is received, the rwlock writer cannot proceed until the last reader completes, and subsequent readers cannot proceed until the writer completes. However, these subsequent readers are guaranteed to see the new value, as indicated by the green shading of the rightmost boxes. In contrast, RCU readers and updaters do not block each other, which permits the RCU readers to see the updated values sooner. Of course, because their execution overlaps that of the RCU updater, *all* of the RCU readers might well see updated values, including the three readers that started before the update. Nevertheless only the green-shaded rightmost RCU readers are *guaranteed* to see the updated values.

Reader-writer locking and RCU simply provide different guarantees. With reader-writer locking, any reader that begins after the writer begins is guaranteed to see new values, and any reader that attempts to begin while the writer is spinning might or might not see new values, depending on the reader/writer preference of the rwlock implementation in question. In contrast, with RCU, any reader that begins after the updater completes is guaranteed to see new values, and any reader that completes after the updater begins might or might not see new values, depending on timing.

The key point here is that, although reader-writer locking does indeed guarantee consistency within the confines of the computer system, there are situations where this consistency comes at the price of increased *inconsistency* with the outside world. In other words, reader-writer locking obtains internal consistency at the price of silently stale data with respect to the outside world.

Nevertheless, there are situations where inconsistency and stale data within the confines of the system cannot be tolerated. Fortunately, there are a number of approaches that avoid inconsistency and stale data [McK04, ACMS03], and some methods based on reference counting are discussed in Section 9.2.

Low-Priority RCU Readers Can Block High-Priority Reclaimers In Realtime RCU [GMTW08] or SRCU [McK06], a preempted reader will prevent a grace period from completing, even if a high-priority task is blocked waiting for that grace period to complete. Realtime RCU can avoid this problem by substituting `call_rcu()` for `synchronize_rcu()` or by using RCU priority boosting [McK07d, GMTW08], which is still in experimental status as of early 2008. It might become necessary to augment SRCU and QRCU with priority boosting, but not before a clear real-world need is demonstrated.

RCU Grace Periods Extend for Many Milliseconds With the exception of userspace RCU [Des09b, MDJ13c],

www.dbooks.org

expedited grace periods, and several of the "toy" RCU implementations described in Appendix B, RCU grace periods extend milliseconds. Although there are a number of techniques to render such long delays harmless, including use of the asynchronous interfaces where available (`call_rcu()` and `call_rcu_bh()`), this situation is a major reason for the rule of thumb that RCU be used in read-mostly situations.

Code: Reader-Writer Locking vs. RCU Code In the best case, the conversion from reader-writer locking to RCU is quite simple, as shown in Listings 9.16, 9.17, and 9.18, all taken from Wikipedia [MPA+06].

However, the transformation is not always this straightforward. This is because neither the `spin_lock()` nor the `synchronize_rcu()` in Listing 9.18 exclude the readers in Listing 9.17. First, the `spin_lock()` does not interact in any way with `rcu_read_lock()` and `rcu_read_unlock()`, thus not excluding them. Second, although both `write_lock()` and `synchronize_rcu()` wait for pre-existing readers, only `write_lock()` prevents subsequent readers from commencing.[17] Thus, `synchronize_rcu()` cannot exclude readers. It is therefore surprising that a great many situations using reader-writer locking can be easily converted to RCU.

More-elaborate cases of replacing reader-writer locking with RCU may be found elsewhere [Bro15a, Bro15b].

Semantics: Reader-Writer Locking vs. RCU Semantics Reader-writer locking semantics can be roughly and informally summarized by the following three temporal constraints:

1. Write-side acquisitions wait for any read-holders to release the lock.
2. Writer-side acquisitions wait for any write-holder to release the lock.
3. Read-side acquisitions wait for any write-holder to release the lock.

RCU dispenses entirely with constraint #3 and weakens the other two as follows:

1. Writers wait for any pre-existing read-holders before progressing to the destructive phase of their update (usually the freeing of memory).
2. Writers synchronize with each other as needed.

[17] Kudos to whoever pointed this out to Paul.

It is of course this weakening that permits RCU implementations to attain excellent performance and scalability. RCU use cases compensate for this weakening in a surprisingly large number of ways, but most commonly by imposing spatial constraints:

1. New data is placed in newly allocated memory.
2. Old data is freed, but only after:
   (a) That data has been unlinked so as to be inaccessible to later readers, and
   (b) A subsequent RCU grace period has elapsed.

In short, RCU attains its read-side performance and scalability by constructing semantics based on combined temporal and spatial constraints.

9.5.4.3 RCU is a Restricted Reference-Counting Mechanism

Because grace periods are not allowed to complete while there is an RCU read-side critical section in progress, the RCU read-side primitives may be used as a restricted reference-counting mechanism. For example, consider the following code fragment:

```
1 rcu_read_lock();  /* acquire reference. */
2 p = rcu_dereference(head);
3 /* do something with p. */
4 rcu_read_unlock();  /* release reference. */
```

The `rcu_read_lock()` primitive can be thought of as acquiring a reference to `p`, because a grace period starting after the `rcu_dereference()` assigns to `p` cannot possibly end until after we reach the matching `rcu_read_unlock()`. This reference-counting scheme is restricted in that we are not allowed to block in RCU read-side critical sections, nor are we permitted to hand off an RCU read-side critical section from one task to another.

Regardless of these restrictions, the following code can safely delete `p`:

```
1 spin_lock(&mylock);
2 p = head;
3 rcu_assign_pointer(head, NULL);
4 spin_unlock(&mylock);
5 /* Wait for all references to be released. */
6 synchronize_rcu();
7 kfree(p);
```

The assignment to `head` prevents any future references to `p` from being acquired, and the `synchronize_rcu()` waits for any previously acquired references to be released.

Listing 9.16: Converting Reader-Writer Locking to RCU: Data

```
1 struct el {
2   struct list_head lp;
3   long key;
4   spinlock_t mutex;
5   int data;
6   /* Other data fields */
7 };
8 DEFINE_RWLOCK(listmutex);
9 LIST_HEAD(head);
```

```
1 struct el {
2   struct list_head lp;
3   long key;
4   spinlock_t mutex;
5   int data;
6   /* Other data fields */
7 };
8 DEFINE_SPINLOCK(listmutex);
9 LIST_HEAD(head);
```

Listing 9.17: Converting Reader-Writer Locking to RCU: Search

```
1 int search(long key, int *result)
2 {
3   struct el *p;
4
5   read_lock(&listmutex);
6   list_for_each_entry(p, &head, lp) {
7     if (p->key == key) {
8       *result = p->data;
9       read_unlock(&listmutex);
10      return 1;
11    }
12  }
13  read_unlock(&listmutex);
14  return 0;
15 }
```

```
1 int search(long key, int *result)
2 {
3   struct el *p;
4
5   rcu_read_lock();
6   list_for_each_entry_rcu(p, &head, lp) {
7     if (p->key == key) {
8       *result = p->data;
9       rcu_read_unlock();
10      return 1;
11    }
12  }
13  rcu_read_unlock();
14  return 0;
15 }
```

Listing 9.18: Converting Reader-Writer Locking to RCU: Deletion

```
1 int delete(long key)
2 {
3   struct el *p;
4
5   write_lock(&listmutex);
6   list_for_each_entry(p, &head, lp) {
7     if (p->key == key) {
8       list_del(&p->lp);
9       write_unlock(&listmutex);
10      kfree(p);
11      return 1;
12    }
13  }
14  write_unlock(&listmutex);
15  return 0;
16 }
```

```
1 int delete(long key)
2 {
3   struct el *p;
4
5   spin_lock(&listmutex);
6   list_for_each_entry(p, &head, lp) {
7     if (p->key == key) {
8       list_del_rcu(&p->lp);
9       spin_unlock(&listmutex);
10      synchronize_rcu();
11      kfree(p);
12      return 1;
13    }
14  }
15  spin_unlock(&listmutex);
16  return 0;
17 }
```


www.dbooks.org

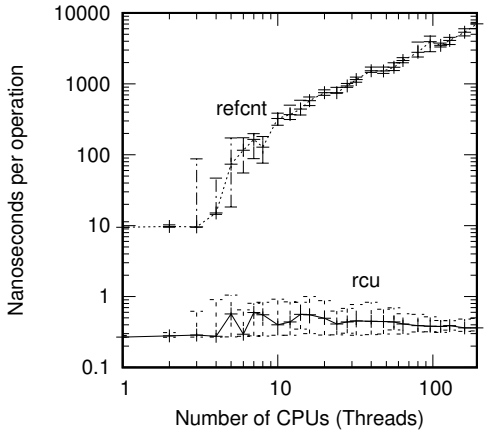


Figure 9.27: Performance of RCU vs. Reference Counting

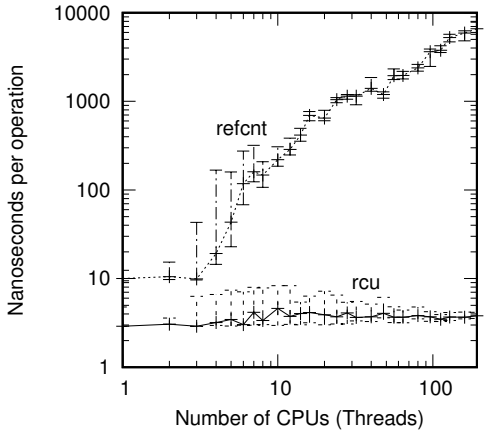


Figure 9.28: Performance of Preemptible RCU vs. Reference Counting

Quick Quiz 9.54: But wait! This is exactly the same code that might be used when thinking of RCU as a replacement for reader-writer locking! What gives? ■

Of course, RCU can also be combined with traditional reference counting, as discussed in Section 13.2.

But why bother? Again, part of the answer is performance, as shown in Figures 9.27 and 9.28, again showing data taken on a 448-CPU 2.1 GHz Intel x86 system for non-preemptible and preemptible Linux-kernel RCU, respectively. Non-preemptible RCU's advantage over reference counting ranges from more than an order of magnitude at one CPU up to about four orders of magnitude at 192 CPUs. Preemptible RCU's advantage ranges

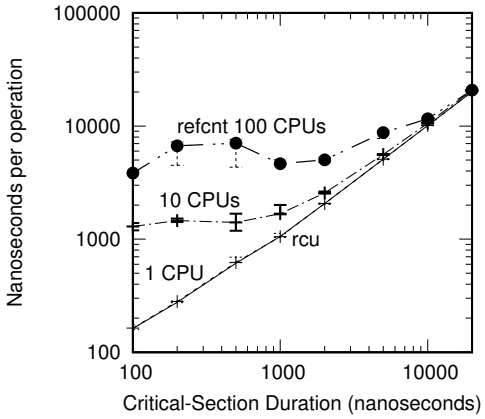


Figure 9.29: Response Time of RCU vs. Reference Counting, 192 CPUs

from about a factor of three at one CPU up to about three orders of magnitude at 192 CPUs.

However, as with reader-writer locking, the performance advantages of RCU are most pronounced for short-duration critical sections and for large numbers of CPUs, as shown in Figure 9.29 for the same system. In addition, as with reader-writer locking, many system calls (and thus any RCU read-side critical sections that they contain) complete in a few microseconds.

However, the restrictions that go with RCU can be quite onerous. For example, in many cases, the prohibition against sleeping while in an RCU read-side critical section would defeat the entire purpose. The next section looks at ways of addressing this problem, while also reducing the complexity of traditional reference counting, at least in some cases.[18]

9.5.4.4 RCU is a Bulk Reference-Counting Mechanism

As noted in the preceding section, traditional reference counters are usually associated with a specific data structure, or perhaps a specific group of data structures. However, maintaining a single global reference counter for a large variety of data structures typically results in bouncing the cache line containing the reference count. Such cache-line bouncing can severely degrade performance.

In contrast, RCU's lightweight read-side primitives permit extremely frequent read-side usage with negligible performance degradation, permitting RCU to be used

[18] Other cases might be better served by the hazard pointers mechanism described in Section 9.3.

as a "bulk reference-counting" mechanism with little or no performance penalty. Situations where a reference must be held by a single task across a section of code that blocks may be accommodated with Sleepable RCU (SRCU) [McK06]. This fails to cover the not-uncommon situation where a reference is "passed" from one task to another, for example, when a reference is acquired when starting an I/O and released in the corresponding completion interrupt handler. (In principle, this could be handled by the SRCU implementation, but in practice, it is not yet clear whether this is a good tradeoff.)

Of course, SRCU brings restrictions of its own, namely that the return value from `srcu_read_lock()` be passed into the corresponding `srcu_read_unlock()`, and that no SRCU primitives be invoked from hardware interrupt handlers or from non-maskable interrupt (NMI) handlers. The jury is still out as to how much of a problem is presented by these restrictions, and as to how they can best be handled.

9.5.4.5 RCU is a Poor Man's Garbage Collector

A not-uncommon exclamation made by people first learning about RCU is "RCU is sort of like a garbage collector!" This exclamation has a large grain of truth, but it can also be misleading.

Perhaps the best way to think of the relationship between RCU and automatic garbage collectors (GCs) is that RCU resembles a GC in that the *timing* of collection is automatically determined, but that RCU differs from a GC in that: (1) the programmer must manually indicate when a given data structure is eligible to be collected, and (2) the programmer must manually mark the RCU read-side critical sections where references might be held.

Despite these differences, the resemblance does go quite deep. In fact, the first RCU-like mechanism I am aware of used a reference-count-based garbage collector to handle the grace periods [KL80], and the connection between RCU and garbage collection has been noted more recently [SWS16]. Nevertheless, a better way of thinking of RCU is described in the following section.

9.5.4.6 RCU is an MVCC

RCU can also be thought of as a simplified multi-version concurrency control (MVCC) mechanism with weak consistency criteria. The multi-version aspects were touched upon in Section 9.5.2.3. However, in its native form, RCU provides version consistency only within a given RCU-protected data element.

Listing 9.19: Existence Guarantees Enable Per-Element Locking

```
1  int delete(int key)
2  {
3    struct element *p;
4    int b;
5
6    b = hashfunction(key);
7    rcu_read_lock();
8    p = rcu_dereference(hashtable[b]);
9    if (p == NULL || p->key != key) {
10     rcu_read_unlock();
11     return 0;
12   }
13   spin_lock(&p->lock);
14   if (hashtable[b] == p && p->key == key) {
15     rcu_read_unlock();
16     rcu_assign_pointer(hashtable[b], NULL);
17     spin_unlock(&p->lock);
18     synchronize_rcu();
19     kfree(p);
20     return 1;
21   }
22   spin_unlock(&p->lock);
23   rcu_read_unlock();
24   return 0;
25 }
```

However, a number of techniques can be used to restore version consistency at a higher level, for example, using sequence locking (see Section 13.4.1) or imposing additional levels of indirection (see Section 13.5.4).

9.5.4.7 RCU Provides Existence Guarantees

Gamsa et al. [GKAS99] discuss existence guarantees and describe how a mechanism resembling RCU can be used to provide these existence guarantees (see Section 5 on page 7 of the PDF), and Section 7.4 discusses how to guarantee existence via locking, along with the ensuing disadvantages of doing so. The effect is that if any RCU-protected data element is accessed within an RCU read-side critical section, that data element is guaranteed to remain in existence for the duration of that RCU read-side critical section.

Listing 9.19 demonstrates how RCU-based existence guarantees can enable per-element locking via a function that deletes an element from a hash table. Line 6 computes a hash function, and line 7 enters an RCU read-side critical section. If line 9 finds that the corresponding bucket of the hash table is empty or that the element present is not the one we wish to delete, then line 10 exits the RCU read-side critical section and line 11 indicates failure.

Quick Quiz 9.55: What if the element we need to delete is not the first element of the list on line 9 of Listing 9.19? ■

www.dbooks.org

Otherwise, line 13 acquires the update-side spinlock, and line 14 then checks that the element is still the one that we want. If so, line 15 leaves the RCU read-side critical section, line 16 removes it from the table, line 17 releases the lock, line 18 waits for all pre-existing RCU read-side critical sections to complete, line 19 frees the newly removed element, and line 20 indicates success. If the element is no longer the one we want, line 22 releases the lock, line 23 leaves the RCU read-side critical section, and line 24 indicates failure to delete the specified key.

Quick Quiz 9.56: Why is it OK to exit the RCU read-side critical section on line 15 of Listing 9.19 before releasing the lock on line 17? ■

Quick Quiz 9.57: Why not exit the RCU read-side critical section on line 23 of Listing 9.19 before releasing the lock on line 22? ■

Alert readers will recognize this as only a slight variation on the original "RCU is a way of waiting for things to finish" theme, which is addressed in Section 9.5.4.9. They might also note the deadlock-immunity advantages over the lock-based existence guarantees discussed in Section 7.4.

9.5.4.8 RCU Provides Type-Safe Memory

A number of lockless algorithms do not require that a given data element keep the same identity through a given RCU read-side critical section referencing it—but only if that data element retains the same type. In other words, these lockless algorithms can tolerate a given data element being freed and reallocated as the same type of structure while they are referencing it, but must prohibit a change in type. This guarantee, called "type-safe memory" in academic literature [GC96], is weaker than the existence guarantees in the previous section, and is therefore quite a bit harder to work with. Type-safe memory algorithms in the Linux kernel make use of slab caches, specially marking these caches with `SLAB_TYPESAFE_BY_RCU` so that RCU is used when returning a freed-up slab to system memory. This use of RCU guarantees that any in-use element of such a slab will remain in that slab, thus retaining its type, for the duration of any pre-existing RCU read-side critical sections.

Quick Quiz 9.58: But what if there is an arbitrarily long series of RCU read-side critical sections in multiple threads, so that at any point in time there is at least one thread in the system executing in an RCU read-side critical section? Wouldn't that prevent any data from a `SLAB_TYPESAFE_BY_RCU` slab ever being returned to the system, possibly resulting in OOM events? ■

It is important to note that `SLAB_TYPESAFE_BY_RCU` will *in no way* prevent `kmem_cache_alloc()` from immediately reallocating memory that was just now freed via `kmem_cache_free()`! In fact, the `SLAB_TYPESAFE_BY_RCU`-protected data structure just returned by `rcu_dereference` might be freed and reallocated an arbitrarily large number of times, even when under the protection of `rcu_read_lock()`. Instead, `SLAB_TYPESAFE_BY_RCU` operates by preventing `kmem_cache_free()` from returning a completely freed-up slab of data structures to the system until after an RCU grace period elapses. In short, although a given RCU read-side critical section might see a given `SLAB_TYPESAFE_BY_RCU` data element being freed and reallocated arbitrarily often, the element's type is guaranteed not to change until that critical section has completed.

These algorithms therefore typically use a validation step that checks to make sure that the newly referenced data structure really is the one that was requested [LS86, Section 2.5]. These validation checks require that portions of the data structure remain untouched by the free-reallocate process. Such validation checks are usually very hard to get right, and can hide subtle and difficult bugs.

Therefore, although type-safety-based lockless algorithms can be extremely helpful in a very few difficult situations, you should instead use existence guarantees where possible. Simpler is after all almost always better!

9.5.4.9 RCU is a Way of Waiting for Things to Finish

As noted in Section 9.5.2 an important component of RCU is a way of waiting for RCU readers to finish. One of RCU's great strength is that it allows you to wait for each of thousands of different things to finish without having to explicitly track each and every one of them, and without having to worry about the performance degradation, scalability limitations, complex deadlock scenarios, and memory-leak hazards that are inherent in schemes that use explicit tracking.

In this section, we will show how `synchronize_sched()`'s read-side counterparts (which include anything that disables preemption, along with hardware operations and primitives that disable interrupts) permit you to implement interactions with non-maskable interrupt (NMI) handlers that would be quite difficult if using locking. This approach has been called "Pure RCU" [McK04], and it is used in a number of places in the Linux kernel.

Listing 9.20: Using RCU to Wait for NMIs to Finish

```
1 struct profile_buffer {
2   long size;
3   atomic_t entry[0];
4 };
5 static struct profile_buffer *buf = NULL;
6
7 void nmi_profile(unsigned long pcvalue)
8 {
9   struct profile_buffer *p = rcu_dereference(buf);
10
11  if (p == NULL)
12    return;
13  if (pcvalue >= p->size)
14    return;
15  atomic_inc(&p->entry[pcvalue]);
16 }
17
18 void nmi_stop(void)
19 {
20  struct profile_buffer *p = buf;
21
22  if (p == NULL)
23    return;
24  rcu_assign_pointer(buf, NULL);
25  synchronize_sched();
26  kfree(p);
27 }
```

The basic form of such "Pure RCU" designs is as follows:

1. Make a change, for example, to the way that the OS reacts to an NMI.

2. Wait for all pre-existing read-side critical sections to completely finish (for example, by using the `synchronize_sched()` primitive). The key observation here is that subsequent RCU read-side critical sections are guaranteed to see whatever change was made.

3. Clean up, for example, return status indicating that the change was successfully made.

The remainder of this section presents example code adapted from the Linux kernel. In this example, the `timer_stop` function uses `synchronize_sched()` to ensure that all in-flight NMI notifications have completed before freeing the associated resources. A simplified version of this code is shown Listing 9.20.

Lines 1–4 define a `profile_buffer` structure, containing a size and an indefinite array of entries. Line 5 defines a pointer to a profile buffer, which is presumably initialized elsewhere to point to a dynamically allocated region of memory.

Lines 7–16 define the `nmi_profile()` function, which is called from within an NMI handler. As such, it cannot be preempted, nor can it be interrupted by a normal interrupts handler, however, it is still subject to delays due to cache misses, ECC errors, and cycle stealing by other hardware threads within the same core. Line 9 gets a local pointer to the profile buffer using the `rcu_dereference()` primitive to ensure memory ordering on DEC Alpha, and lines 11 and 12 exit from this function if there is no profile buffer currently allocated, while lines 13 and 14 exit from this function if the `pcvalue` argument is out of range. Otherwise, line 15 increments the profile-buffer entry indexed by the `pcvalue` argument. Note that storing the size with the buffer guarantees that the range check matches the buffer, even if a large buffer is suddenly replaced by a smaller one.

Lines 18–27 define the `nmi_stop()` function, where the caller is responsible for mutual exclusion (for example, holding the correct lock). Line 20 fetches a pointer to the profile buffer, and lines 22 and 23 exit the function if there is no buffer. Otherwise, line 24 NULLs out the profile-buffer pointer (using the `rcu_assign_pointer()` primitive to maintain memory ordering on weakly ordered machines), and line 25 waits for an RCU Sched grace period to elapse, in particular, waiting for all non-preemptible regions of code, including NMI handlers, to complete. Once execution continues at line 26, we are guaranteed that any instance of `nmi_profile()` that obtained a pointer to the old buffer has returned. It is therefore safe to free the buffer, in this case using the `kfree()` primitive.

Quick Quiz 9.59: Suppose that the `nmi_profile()` function was preemptible. What would need to change to make this example work correctly? ■

In short, RCU makes it easy to dynamically switch among profile buffers (you just *try* doing this efficiently with atomic operations, or at all with locking!). However, RCU is normally used at a higher level of abstraction, as was shown in the previous sections.

9.5.4.10 RCU Usage Summary

At its core, RCU is nothing more nor less than an API that provides:

1. a publish-subscribe mechanism for adding new data,

2. a way of waiting for pre-existing RCU readers to finish, and

3. a discipline of maintaining multiple versions to permit change without harming or unduly delaying concurrent RCU readers.

www.dbooks.org

That said, it is possible to build higher-level constructs on top of RCU, including the reader-writer-locking, reference-counting, and existence-guarantee constructs listed in the earlier sections. Furthermore, I have no doubt that the Linux community will continue to find interesting new uses for RCU, as well as for any of a number of other synchronization primitives.

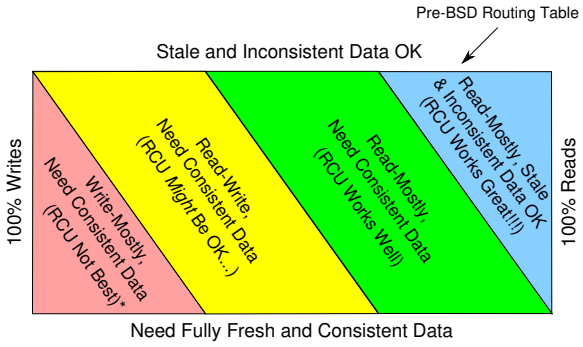


Figure 9.30: RCU Areas of Applicability

In the meantime, Figure 9.30 shows some rough rules of thumb on where RCU is most helpful.

As shown in the blue box at the top of the figure, RCU works best if you have read-mostly data where stale and inconsistent data is permissible (but see below for more information on stale and inconsistent data). The canonical example of this case in the Linux kernel is routing tables. Because it may have taken many seconds or even minutes for the routing updates to propagate across the Internet, the system has been sending packets the wrong way for quite some time. Having some small probability of continuing to send some of them the wrong way for a few more milliseconds is almost never a problem.

If you have a read-mostly workload where consistent data is required, RCU works well, as shown by the green "read-mostly, need consistent data" box. One example of this case is the Linux kernel's mapping from user-level System-V semaphore IDs to the corresponding in-kernel data structures. Semaphores tend to be used far more frequently than they are created and destroyed, so this mapping is read-mostly. However, it would be erroneous to perform a semaphore operation on a semaphore that has already been deleted. This need for consistency is handled by using the lock in the in-kernel semaphore data structure, along with a "deleted" flag that is set when deleting a semaphore. If a user ID maps to an in-kernel data structure with the "deleted" flag set, the data structure is ignored, so that the user ID is flagged as invalid.

Although this requires that the readers acquire a lock for the data structure representing the semaphore itself, it allows them to dispense with locking for the mapping data structure. The readers therefore locklessly traverse the tree used to map from ID to data structure, which in turn greatly improves performance, scalability, and real-time response.

As indicated by the yellow "read-write" box, RCU can also be useful for read-write workloads where consistent data is required, although usually in conjunction with a number of other synchronization primitives. For example, the directory-entry cache in recent Linux kernels uses RCU in conjunction with sequence locks, per-CPU locks, and per-data-structure locks to allow lockless traversal of pathnames in the common case. Although RCU can be very beneficial in this read-write case, such use is often more complex than that of the read-mostly cases.

Finally, as indicated by the red box at the bottom of the figure, update-mostly workloads requiring consistent data are rarely good places to use RCU, though there are some exceptions [DMS+12]. In addition, as noted in Section 9.5.4.8, within the Linux kernel, the `SLAB_TYPESAFE_BY_RCU` slab-allocator flag provides type-safe memory to RCU readers, which can greatly simplify non-blocking synchronization and other lockless algorithms.

In short, RCU is an API that includes a publish-subscribe mechanism for adding new data, a way of waiting for pre-existing RCU readers to finish, and a discipline of maintaining multiple versions to allow updates to avoid harming or unduly delaying concurrent RCU readers. This RCU API is best suited for read-mostly situations, especially if stale and inconsistent data can be tolerated by the application.

9.5.5 RCU Related Work

The known first mention of anything resembling RCU took the form of a bug report from Donald Knuth [Knu73, page 413 of Fundamental Algorithms] against Joseph Weizenbaum's SLIP list-processing facility for FORTRAN [Wei63]. Knuth was justified in reporting the bug, as SLIP had no notion of any sort of grace-period guarantee.

The first known non-bug-report mention of anything resembling RCU appeared in Kung's and Lehman's landmark paper [KL80]. There was some additional use of this technique in academia [ML82, ML84, Lis88, Pug90, And91, PAB+95, CAK+96, RSB+97, GKAS99],

but much of the work in this area was carried out by practitioners [RTY+87, HOS89, Jac93, Joh95, SM95, SM97, SM98, MS98a]. By the year 2000, the initiative had passed to open-source projects, most notably the Linux kernel community [Rus00a, Rus00b, MS01, MAK+01, MSA+02, ACMS03].[19]

However, in the mid 2010s, there was a welcome upsurge in RCU research and development across a number of communities and institutions [Kaa15]. Section 9.5.5.1 describes uses of RCU, Section 9.5.5.2 describes RCU implementations (as well as work that both creates and uses an implementation), and finally, Section 9.5.5.3 describes verification and validation of RCU and its uses.

9.5.5.1 RCU Uses

Phil Howard and Jon Walpole of Portland State University (PSU) have applied RCU to red-black trees [How12, HW11] combined with updates synchronized using software transactional memory. Josh Triplett and Jon Walpole (again of PSU) applied RCU to resizable hash tables [Tri12, TMW11, Cor14c, Cor14d]. Other RCU-protected resizable hash tables have been created by Herbert Xu [Xu10] and by Mathieu Desnoyers [MDJ13a].

Austin Clements, Frans Kaashoek, and Nickolai Zeldovich of MIT created an RCU-optimized balanced binary tree (Bonsai) [CKZ12], and applied this tree to the Linux kernel's VM subsystem in order to reduce read-side contention on the Linux kernel's `mmap_sem`. This work resulted in order-of-magnitude speedups and scalability up to at least 80 CPUs for a microbenchmark featuring large numbers of minor page faults. This is similar to a patch developed earlier by Peter Zijlstra [Zij14], and both were limited by the fact that, at the time, filesystem data structures were not safe for RCU readers. Clements et al. avoided this limitation by optimizing the page-fault path for anonymous pages only. More recently, filesystem data structures have been made safe for RCU readers [Cor10a, Cor11], so perhaps this work can be implemented for all page types, not just anonymous pages—Peter Zijlstra has, in fact, recently prototyped exactly this, and Laurent Dufour has continued work along these lines.

Yandong Mao and Robert Morris of MIT and Eddie Kohler of Harvard University created another RCU-protected tree named Masstree [MKM12] that combines ideas from B+ trees and tries. Although this tree is about 2.5x slower than an RCU-protected hash table, it supports operations on key ranges, unlike hash tables. In addition, Masstree supports efficient storage of objects with long shared key prefixes and, furthermore, provides persistence via logging to mass storage.

The paper notes that Masstree's performance rivals that of memcached, even given that Masstree is persistently storing updates and memcached is not. The paper also compares Masstree's performance to the persistent datastores MongoDB, VoltDB, and Redis, reporting significant performance advantages for Masstree, in some cases exceeding two orders of magnitude. Another paper [TZK+13], by Stephen Tu, Wenting Zheng, Barbara Liskov, and Samuel Madden of MIT and Kohler, applies Masstree to an in-memory database named Silo, achieving 700K transactions per second (42M transactions per minute) on a well-known transaction-processing benchmark. Interestingly enough, Silo guarantees linearizability without incurring the overhead of grace periods while holding locks.

Maya Arbel and Hagit Attiya of Technion took a more rigorous approach [AA14] to an RCU-protected search tree that, like Masstree, allows concurrent updates. This paper includes a proof of correctness, including proof that all operations on this tree are linearizable. Unfortunately, this implementation achieves linearizability by incurring the full latency of grace-period waits while holding locks, which degrades scalability of update-only workloads. One way around this problem is to abandon linearizability [HKLP12, McK14b]), however, Arbel and Attiya instead created an RCU variant that reduces low-end grace-period latency. Of course, nothing comes for free, and this RCU variant appears to hit a scalability limit at about 32 CPUs. Although there is much to be said for dropping linearizability, thus gaining both performance and scalability, it is very good to see academics experimenting with alternative RCU implementations.

9.5.5.2 RCU Implementations

Mathieu Desnoyers created a user-space RCU for use in tracing [Des09b, Des09a, DMS+12], which has seen use in a number of projects [BD13].

Researchers at Charles University in Prague have also been working on RCU implementations, including dissertations by Andrej Podzimek [Pod10] and Adam Hraska [Hra13].

Yujie Liu (Lehigh University), Victor Luchangco (Oracle Labs), and Michael Spear (also Lehigh) [LLS13] pressed scalable non-zero indicators (SNZI) [ELLM07] into service as a grace-period mechanism. The intended

[19] A list of citations with well over 200 entries may be found in `bib/RCU.bib` in the LaTeX source for this book.

www.dbooks.org

use is to implement software transactional memory (see Section 17.2), which imposes linearizability requirements, which in turn seems to limit scalability.

RCU-like mechanisms are also finding their way into Java. Sivaramakrishnan et al. [SZJ12] use an RCU-like mechanism to eliminate the read barriers that are otherwise required when interacting with Java's garbage collector, resulting in significant performance improvements.

Ran Liu, Heng Zhang, and Haibo Chen of Shanghai Jiao Tong University created a specialized variant of RCU that they used for an optimized "passive reader-writer lock" [LZC14], similar to those created by Gautham Shenoy [She06] and Srivatsa Bhat [Bha14]. The Liu et al. paper is interesting from a number of perspectives [McK14e].

Mike Ash posted [Ash15] a description of an RCU-like primitive in Apple's Objective-C runtime. This approach identifies read-side critical sections via designated code ranges, thus qualifying as another method of achieving zero read-side overhead, albeit one that poses some interesting practical challenges for large read-side critical sections that span multiple functions.

Pedro Ramalhete and Andreia Correia [RC15] produced "Poor Man's RCU", which, despite using a pair of reader-writer locks, manages to provide lock-free forward-progress guarantees to readers [MP15a].

Maya Arbel and Adam Morrison [AM15] produced "Predicate RCU", which works hard to reduce grace-period duration in order to efficiently support algorithms that hold update-side locks across grace periods. This results in reduced batching of updates into grace periods and reduced scalability, but does succeed in providing short grace periods.

Quick Quiz 9.60: Why not just drop the lock before waiting for the grace period, or using something like `call_rcu()` instead of waiting for a grace period? ■

Alexander Matveev (MIT), Nir Shavit (MIT and Tel-Aviv University), Pascal Felber (University of Neuchâtel), and Patrick Marlier (also University of Neuchâtel) [MSFM15] produced an RCU-like mechanism that can be thought of as software transactional memory that explicitly marks read-only transactions. Their use cases require holding locks across grace periods, which limits its scalability [MP15a, MP15b]. This appears to be the first academic RCU-related work to make good use of the `rcutorture` test suite, and also the first to have submitted a performance improvement to Linux-kernel RCU, which was accepted into v4.4.

Alexander Matveev's RLU was followed up by MV-RLU from Jaeho Kim et al. [KMK+19]. This work improves scalability over RLU by permitting multiple concurrent updates, by avoiding holding locks across grace periods, and by using asynchronous grace periods, for example, `call_rcu()` instead of `synchronize_rcu()`. This paper also made some interesting performance-evaluation choices that are discussed further in Section 17.2.3.3 on page 17.2.3.3.

Adam Belay et al. created an RCU implementation that guards the data structures used by TCP/IP's address-resolution protocol (ARP) in their IX operating system [BPP+16].

Geoff Romer and Andrew Hunter (both at Google) proposed a cell-based API for RCU protection of singleton data structures for inclusion in the C++ standard [RH18].

Dimitrios Siakavaras et al. have applied HTM and RCU to search trees [SNGK17, SBN+20], Christina Giannoula et al. have used HTM and RCU to color graphs [GGK18], and SeongJae Park et al. have used HTM and RCU to optimize high-contention locking on NUMA systems.

Alex Kogan et al. applied RCU to the construction of range locking for scalable address spaces [KDI20].

9.5.5.3 RCU Validation

In early 2017, it is commonly recognized that almost any bug is a potential security exploit, so validation and verification are first-class concerns.

Researchers at Stony Brook University have produced an RCU-aware data-race detector [Dug10, Sey12, SRK+11]. Alexey Gotsman of IMDEA, Noam Rinetzky of Tel Aviv University, and Hongseok Yang of the University of Oxford have published a paper [GRY12] expressing the formal semantics of RCU in terms of separation logic, and have continued with other aspects of concurrency.

Joseph Tassarotti (Carnegie-Mellon University), Derek Dreyer (Max Planck Institute for Software Systems), and Viktor Vafeiadis (also MPI-SWS) [TDV15] produced a manual formal proof of correctness of the quiescent-state-based reclamation (QSBR) variant of userspace RCU [Des09b, DMS+12]. Lihao Liang (University of Oxford), Paul E. McKenney (IBM), Daniel Kroening, and Tom Melham (both also Oxford) [LMKM16] used the C bounded model checker (CBMC) [CKL04] to produce a mechanical proof of correctness of a significant portion of Linux-kernel Tree RCU. Lance Roy [Roy17] used CBMC to produce a similar proof of correctness for a significant portion of Linux-kernel sleepable RCU (SRCU) [McK06]. Finally, Michalis Kokologiannakis and

Konstantinos Sagonas (National Technical University of Athens) [KS17a, KS19] used the Nighugg tool [LSLK14] to produce a mechanical proof of correctness of a somewhat larger portion of Linux-kernel Tree RCU.

None of these efforts located any bugs other than bugs injected into RCU specifically to test the verification tools. In contrast, Alex Groce (Oregon State University), Iftekhar Ahmed, Carlos Jensen (both also OSU), and Paul E. McKenney (IBM) [GAJM15] automatically mutated Linux-kernel RCU's source code to test the coverage of the `rcutorture` test suite. The effort found several holes in this suite's coverage, one of which was hiding a real bug (since fixed) in Tiny RCU.

With some luck, all of this validation work will eventually result in more and better tools for validating concurrent code.

9.5.6 RCU Exercises

This section is organized as a series of Quick Quizzes that invite you to apply RCU to a number of examples earlier in this book. The answer to each Quick Quiz gives some hints, and also contains a pointer to a later section where the solution is explained at length. The `rcu_read_lock()`, `rcu_read_unlock()`, `rcu_dereference()`, `rcu_assign_pointer()`, and `synchronize_rcu()` primitives should suffice for most of these exercises.

Quick Quiz 9.61: The statistical-counter implementation shown in Listing 5.4 (`count_end.c`) used a global lock to guard the summation in `read_count()`, which resulted in poor performance and negative scalability. How could you use RCU to provide `read_count()` with excellent performance and good scalability. (Keep in mind that `read_count()`'s scalability will necessarily be limited by its need to scan all threads' counters.) ■

Quick Quiz 9.62: Section 5.4.6 showed a fanciful pair of code fragments that dealt with counting I/O accesses to removable devices. These code fragments suffered from high overhead on the fastpath (starting an I/O) due to the need to acquire a reader-writer lock. How would you use RCU to provide excellent performance and scalability? (Keep in mind that the performance of the common-case first code fragment that does I/O accesses is much more important than that of the device-removal code fragment.) ■

9.6 Which to Choose?

> Choose always the way that seems the best, however rough it may be; custom will soon render it easy and agreeable.
>
> *Pythagoras*

Section 9.6.1 provides a high-level overview and then Section 9.6.2 provides a more detailed view of the differences between the deferred-processing techniques presented in this chapter. This discussion assumes a linked data structure that is large enough that readers do not hold references from one traversal to another, and where elements might be added to and removed from the structure at any location and at any time. Section 9.6.3 then points out a few publicly visible production uses of hazard pointers, sequence locking, and RCU. This discussion should help you to make an informed choice between these techniques.

9.6.1 Which to Choose? (Overview)

Table 9.6 shows a few high-level properties that distinguish the deferred-reclamation techniques from one another.

The "Readers" row summarizes the results presented in Figure 9.22, which shows that all but reference counting are enjoy reasonably fast and scalable readers.

The "Number of Protected Objects" row evaluates each technique's need for external storage with which to record reader protection. RCU relies on quiescent states, and thus needs no storage to represent readers, whether within or outside of the object. Reference counting can use a single integer within each object in the structure, and no additional storage is required. Hazard pointers require external-to-object pointers be provisioned, and that there be sufficient pointers to handle the maximum number of objects that a given CPU or thread might need to reference simultaneously. Of course, sequence locks provides no pointer-traversal protection, which is why it is normally used on static data.

Quick Quiz 9.63: Why can't users dynamically allocate the hazard pointers as they are needed? ■

The "Duration of Protection" describes constraints (if any) on how long a period of time a user may protect a given object. Reference counting and hazard pointers can both protect objects for extended time periods with no untoward side effects, but maintaining an RCU reference to even one object prevents all other RCU from being freed. RCU readers must therefore be relatively short in order to

www.dbooks.org

Table 9.6: Which Deferred Technique to Choose? (Overview)

Property	Reference Counting	Hazard Pointers	Sequence Locks	RCU
Readers	Slow and unscalable	Fast and scalable	Fast and scalable	Fast and scalable
Number of Protected Objects	Scalable	Unscalable	No protection	Scalable
Duration of Protection	Can be long	Can be long	No protection	User must bound duration
Need for Traversal Retries	If race with object deletion	If race with object deletion	If race with any update	Never

avoid running the system out of memory, which special-purpose implementations such as SRCU, Tasks RCU, and Tasks Trace RCU being exceptions to this rule. Again, sequence locks provide no pointer-traversal protection, which is why it is normally used on static data.

The "Need for Traversal Retries" row tells whether a new reference to a given object may be acquired unconditionally, as it can with RCU, or whether the reference acquisition can fail, resulting in a retry operation, which is the case for reference counting, hazard pointers, and sequence locks. In the case of reference counting and hazard pointers, retries are only required if an attempt to acquire a reference to a given object while that object is in the process of being deleted, a topic covered in more detail in the next section. Sequence locking must of course retry its critical section should it run concurrently with any update.

Quick Quiz 9.64: But don't Linux-kernel `kref` reference counters allow guaranteed unconditional reference acquisition? ■

Of course, different rows will have different levels of importance in different situations. For example, if your current code is having read-side scalability problems with hazard pointers, then it does not matter that hazard pointers can require retrying reference acquisition because your current code already handles this. Similarly, if response-time considerations already limit the duration of reader traversals, as is often the case in kernels and low-level applications, then it does not matter that RCU has duration-limit requirements because your code already meets them. In the same vein, if readers must already write to the objects that they are traversing, the read-side overhead of reference counters might not be so important. Of course, if the data to be protected is in statically allocated variables, then sequence locking's inability to protect pointers is irrelevant.

Finally, there is some work on dynamically switching between hazard pointers and RCU based on dynamic sampling of delays [BGHZ16]. This defers the choice between hazard pointers and RCU to runtime, and delegates responsibility for the decision to the software.

Nevertheless, this table should be of great help when choosing between these techniques. But those wishing more detail should continue on to the next section.

9.6.2 Which to Choose? (Details)

Table 9.7 provides more-detailed rules of thumb that can help you choose among the four deferred-processing techniques presented in this chapter.

As shown in the "Existence Guarantee" row, if you need existence guarantees for linked data elements, you must use reference counting, hazard pointers, or RCU. Sequence locks do not provide existence guarantees, instead providing detection of updates, retrying any read-side critical sections that do encounter an update.

Of course, as shown in the "Updates and Readers Progress Concurrently" row, this detection of updates implies that sequence locking does not permit updaters and readers to make forward progress concurrently. After all, preventing such forward progress is the whole point of using sequence locking in the first place! This situation points the way to using sequence locking in conjunction with reference counting, hazard pointers, or RCU in order to provide both existence guarantees and update detection. In fact, the Linux kernel combines RCU and sequence locking in this manner during pathname lookup.

The "Contention Among Readers", "Reader Per-Critical-Section Overhead", and "Reader Per-Object Traversal Overhead" rows give a rough sense of the read-side overhead of these techniques. The overhead of reference counting can be quite large, with contention among readers along with a fully ordered read-modify-write atomic operation required for each and every object traversed.

Table 9.7: Which Deferred Technique to Choose? (Details)

Property	Reference Counting	Hazard Pointers	Sequence Locks	RCU
Existence Guarantees	Complex	Yes	No	Yes
Updates and Readers Progress Concurrently	Yes	Yes	No	Yes
Contention Among Readers	High	None	None	None
Reader Per-Critical-Section Overhead	N/A	N/A	Two `smp_mb()`	Ranges from none to two `smp_mb()`
Reader Per-Object Traversal Overhead	Read-modify-write atomic operations, memory-barrier instructions, and cache misses	`smp_mb()`	None, but unsafe	None (volatile accesses)
Reader Forward Progress Guarantee	Lock free	Lock free	Blocking	Bounded wait free
Reader Reference Acquisition	Can fail (conditional)	Can fail (conditional)	Unsafe	Cannot fail (unconditional)
Memory Footprint	Bounded	Bounded	Bounded	Unbounded
Reclamation Forward Progress	Lock free	Lock free	N/A	Blocking
Automatic Reclamation	Yes	Use Case	N/A	Use Case
Lines of Code	94	79	79	73

Hazard pointers incur the overhead of a memory barrier for each data element traversed, and sequence locks incur the overhead of a pair of memory barriers for each attempt to execute the critical section. The overhead of RCU implementations vary from nothing to that of a pair of memory barriers for each read-side critical section, thus providing RCU with the best performance, particularly for read-side critical sections that traverse many data elements. Of course, the read-side overhead of all deferred-processing variants can be reduced by batching, so that each read-side operation covers more data.

Quick Quiz 9.65: But didn't the answer to one of the quick quizzes in Section 9.3 say that pairwise asymmetric barriers could eliminate the read-side `smp_mb()` from hazard pointers? ■

The "Reader Forward Progress Guarantee" row shows that only RCU has a bounded wait-free forward-progress guarantee, which means that it can carry out a finite traversal by executing a bounded number of instructions.

The "Reader Reference Acquisition" rows indicates that only RCU is capable of unconditionally acquiring references. The entry for sequence locks is "Unsafe" because, again, sequence locks detect updates rather than acquiring references. Reference counting and hazard pointers both require that traversals be restarted from the beginning if a given acquisition fails. To see this, consider a linked list containing objects A, B, C, and D, in that order, and the following series of events:

1. A reader acquires a reference to object B.
2. An updater removes object B, but refrains from freeing it because the reader holds a reference. The list now contains objects A, C, and D, and object B's `->next` pointer is set to `HAZPTR_POISON`.
3. The updater removes object C, so that the list now contains objects A and D. Because there is no reference to object C, it is immediately freed.
4. The reader tries to advance to the successor of the object following the now-removed object B, but the poisoned `->next` pointer prevents this. Which is a good thing, because object B's `->next` pointer would otherwise point to the freelist.
5. The reader must therefore restart its traversal from the head of the list.

Thus, when failing to acquire a reference, a hazard-pointer or reference-counter traversal must restart that traversal from the beginning. In the case of nested linked data structures, for example, a tree containing linked

www.dbooks.org

lists, the traversal must be restarted from the outermost data structure. This situation gives RCU a significant ease-of-use advantage.

However, RCU's ease-of-use advantage does not come for free, as can be seen in the "Memory Footprint" row. RCU's support of unconditional reference acquisition means that it must avoid freeing any object reachable by a given RCU reader until that reader completes. RCU therefore has an unbounded memory footprint, at least unless updates are throttled. In contrast, reference counting and hazard pointers need to retain only those data elements actually referenced by concurrent readers.

This tension between memory footprint and acquisition failures is sometimes resolved within the Linux kernel by combining use of RCU and reference counters. RCU is used for short-lived references, which means that RCU read-side critical sections can be short. These short RCU read-side critical sections in turn mean that the corresponding RCU grace periods can also be short, which limits the memory footprint. For the few data elements that need longer-lived references, reference counting is used. This means that the complexity of reference-acquisition failure only needs to be dealt with for those few data elements: The bulk of the reference acquisitions are unconditional, courtesy of RCU. See Section 13.2 for more information on combining reference counting with other synchronization mechanisms.

The "Reclamation Forward Progress" row shows that hazard pointers can provide non-blocking updates [Mic04, HLM02]. Reference counting might or might not, depending on the implementation. However, sequence locking cannot provide non-blocking updates, courtesy of its update-side lock. RCU updaters must wait on readers, which also rules out fully non-blocking updates. However, there are situations in which the only blocking operation is a wait to free memory, which results in a situation that, for many purposes, is as good as non-blocking [DMS+12].

As shown in the "Automatic Reclamation" row, only reference counting can automate freeing of memory, and even then only for non-cyclic data structures. Certain use cases for hazard pointers and RCU can provide automatic reclamation using *link counts*, which can be thought of as reference counts, but applying only to incoming links from other parts of the data structure [Mic18].

Finally, the "Lines of Code" row shows the size of the Pre-BSD Routing Table implementations, giving a rough idea of relative ease of use. That said, it is important to note that the reference-counting and sequence-locking implementations are buggy, and that a correct reference-counting implementation is considerably more complex [Val95, MS95]. For its part, a correct sequence-locking implementation requires the addition of some other synchronization mechanism, for example, hazard pointers or RCU, so that sequence locking detects concurrent updates and the other mechanism provides safe reference acquisition.

As more experience is gained using these techniques, both separately and in combination, the rules of thumb laid out in this section will need to be refined. However, this section does reflect the current state of the art.

9.6.3 Which to Choose? (Production Use)

This section points out a few publicly visible production uses of hazard pointers, sequence locking, and RCU. Reference counting is omitted, not because it is unimportant, but rather because it is not only used pervasively, but heavily documented in textbooks going back a half century. One of the hoped-for benefits of listing production uses of these other techniques is to provide examples to study—or to find bugs in, as the case may be.[20]

9.6.3.1 Production Uses of Hazard Pointers

In 2010, Keith Bostic added hazard pointers to WiredTiger [Bos10]. MongoDB 3.0, released in 2015, included WiredTiger and thus hazard pointers.

In 2011, Samy Al Bahra added hazard pointers to the Concurrency Kit library [Bah11b].

In 2014, Maxim Khizhinsky added hazard pointers to libcds [Khi14].

In 2015, David Gwynne introduced shared reference pointers, a form of hazard pointers, to OpenBSD [Gwy15].

In 2017–2018, the Rust-language `arc-swap` [Van18] and `conc` [cut17] crates rolled their own implementations of hazard pointers.

In 2018, Maged Michael added hazard pointers to Facebook's Folly library [Mic18], where it is used heavily.

9.6.3.2 Production Uses of Sequence Locking

The Linux kernel added sequence locking to v2.5.60 in 2003 [Cor03], having been generalized from an ad-hoc technique used in x86's implementation of the `gettimeofday()` system call.

[20] Kudos to Mathias Stearn, Matt Wilson, David Goldblatt, LiveJournal user fanf, Nadav Har'El, Avi Kivity, Dmitry Vyukov, Raul Guitterez S., Twitter user @peo3, Paolo Bonzini, and Thomas Monjalon for locating a great many of these use cases.

In 2011, Samy Al Bahra added sequence locking to the Concurrency Kit library [Bah11c].

Paolo Bonzini added a simple sequence-lock to the QEMU emulator in 2013 [Bon13].

Alexis Menard abstracted a sequence-lock implementation in Chromium in 2016 [Men16].

A simple sequence locking implementation was added to `jemalloc()` in 2018 [Gol18a]. The eigen library also has a special-purpose queue that is managed by a mechanism resembling sequence locking.

9.6.3.3 Production Uses of RCU

IBM's VM/XA is adopted passive serialization, a mechanism similar to RCU, some time in the 1980s [HOS89].

DYNIX/ptx adopted RCU in 1993 [MS98a, SM95].

The Linux kernel adopted Dipankar Sarma's implementation of RCU in 2002 [Tor02].

The userspace RCU project started in 2009 [Des09b].

The Knot DNS project started using the userspace RCU library in 2010 [Slo10]. That same year, the OSv kernel added an RCU implementation [Kiv13], later adding an RCU-protected linked list [Kiv14b] and an RCU-protected hash table [Kiv14a].

In 2011, Samy Al Bahra added epochs (a form of RCU [Fra04, FH07]) to the Concurrency Kit library [Bah11a].

NetBSD began using the aforementioned passive serialization with v6.0 in 2012 [The12a]. Among other things, passive serialization is used in NetBSD packet filter (NPF) [Ras14].

Paolo Bonzini added RCU support to the QEMU emulator in 2015 via a friendly fork of the userspace RCU library [BD13, Bon15].

In 2015, Maxim Khizhinsky added RCU to libcds [Khi15].

Mindaugas Rasiukevicius implemented libqsbr in 2016, which features QSBR and epoch-based reclamation (EBR) [Ras16], both of which are types of implementations of RCU.

Sheth et al. [SWS16] demonstrated the value of leveraging Go's garbage collector to provide RCU-like functionality, and the Go programming language provides a `Value` type that can provide this functionality.[21]

Matt Klein describes an RCU-like mechanism that is used in the Envoy Proxy [Kle17].

[21] See `https://golang.org/pkg/sync/atomic/#Value`, particularly the "Example (ReadMostly)".

Honnappa Nagarahalli added an RCU library to the Data Plane Development Kit (DPDK) in 2018 [Nag18].

Stjepan Glavina merged an epoch-based RCU implementation into the `crossbeam` set of concurrency-support "crates" for the Rust language [Gla18].

Finally, any garbage-collected concurrent language (not just Go!) gets the update side of an RCU implementation at zero incremental cost.

9.6.3.4 Summary of Production Uses

Perhaps the time will come when sequence locking, hazard pointers, and RCU are all as heavily used and as well known as are reference counters. Until that time comes, the current production uses of these mechanisms should help guide the choice of mechanism as well as showing how best to apply each of them.

The next section discusses updates, a ticklish issue for many of the read-mostly mechanisms described in this chapter.

9.7 What About Updates?

> The only thing constant in life is change.
>
> *François de la Rochefoucauld*

The deferred-processing techniques called out in this chapter are most directly applicable to read-mostly situations, which begs the question "But what about updates?" After all, increasing the performance and scalability of readers is all well and good, but it is only natural to also want great performance and scalability for writers.

We have already seen one situation featuring high performance and scalability for writers, namely the counting algorithms surveyed in Chapter 5. These algorithms featured partially partitioned data structures so that updates can operate locally, while the more-expensive reads must sum across the entire data structure. Silas Boyd-Wickhizer has generalized this notion to produce OpLog, which he has applied to Linux-kernel pathname lookup, VM reverse mappings, and the `stat()` system call [BW14].

Another approach, called "Disruptor", is designed for applications that process high-volume streams of input data. The approach is to rely on single-producer-single-consumer FIFO queues, minimizing the need for synchronization [Sut13]. For Java applications, Disruptor also has the virtue of minimizing use of the garbage collector.

www.dbooks.org

And of course, where feasible, fully partitioned or "sharded" systems provide excellent performance and scalability, as noted in Chapter 6.

The next chapter will look at updates in the context of several types of data structures.

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

Linus Torvalds

Chapter 10

Data Structures

Serious discussions of algorithms include time complexity of their data structures [CLRS01]. However, for parallel programs, the time complexity includes concurrency effects because these effects can be overwhelmingly large, as shown in Chapter 3. In other words, a good programmer's data-structure relationships include those aspects related to concurrency.

Section 10.1 presents the motivating application for this chapter's data structures. Chapter 6 showed how partitioning improves scalability, so Section 10.2 discusses partitionable data structures. Chapter 9 described how deferring some actions can greatly improve both performance and scalability, a topic taken up by Section 10.3. Section 10.4 looks at a non-partitionable data structure, splitting it into read-mostly and partitionable portions, which improves both performance and scalability. Because this chapter cannot delve into the details of every concurrent data structure, Section 10.5 surveys a few of the important ones. Although the best performance and scalability results from design rather than after-the-fact micro-optimization, micro-optimization is nevertheless necessary for the absolute best possible performance and scalability, as described in Section 10.6. Finally, Section 10.7 presents a summary of this chapter.

10.1 Motivating Application

The art of doing mathematics consists in finding that special case which contains all the germs of generality.

David Hilbert

We will use the Schrödinger's Zoo application to evaluate performance [McK13]. Schrödinger has a zoo containing a large number of animals, and he would like to track them using an in-memory database with each animal in the zoo represented by a data item in this database. Each animal has a unique name that is used as a key, with a variety of data tracked for each animal.

Births, captures, and purchases result in insertions, while deaths, releases, and sales result in deletions. Because Schrödinger's zoo contains a large quantity of short-lived animals, including mice and insects, the database must handle high update rates. Those interested in Schrödinger's animals can query them, and Schrödinger has noted suspiciously query rates for his cat, so much so that he suspects that his mice might be checking up on their nemesis. Whatever their source, Schrödinger's application must handle high query rates to a single data element.

As we will see, this simple application can be a challenge to concurrent data structures.

10.2 Partitionable Data Structures

Finding a way to live the simple life today is the most complicated task.

Henry A. Courtney, updated

There are a huge number of data structures in use today, so much so that there are multiple textbooks covering them. This section focuses on a single data structure, namely the hash table. This focused approach allows a much deeper investigation of how concurrency interacts with data structures, and also focuses on a data structure that is heavily used in practice. Section 10.2.1 overviews the design, and Section 10.2.2 presents the implementation. Finally, Section 10.2.3 discusses the resulting performance and scalability.

Edition.2-rc11

www.dbooks.org

10.2.1 Hash-Table Design

Chapter 6 emphasized the need to apply partitioning in order to attain respectable performance and scalability, so partitionability must be a first-class criterion when selecting data structures. This criterion is well satisfied by that workhorse of parallelism, the hash table. Hash tables are conceptually simple, consisting of an array of *hash buckets*. A *hash function* maps from a given element's *key* to the hash bucket that this element will be stored in. Each hash bucket therefore heads up a linked list of elements, called a *hash chain*. When properly configured, these hash chains will be quite short, permitting a hash table to access its elements extremely efficiently.

Quick Quiz 10.1: But chained hash tables are but one type of many. Why the focus on chained hash tables? ■

In addition, each bucket has its own lock, so that elements in different buckets of the hash table may be added, deleted, and looked up completely independently. A large hash table with a large number of buckets (and thus locks), with each bucket containing a small number of elements should therefore provide excellent scalability.

10.2.2 Hash-Table Implementation

Listing 10.1 (`hash_bkt.c`) shows a set of data structures used in a simple fixed-sized hash table using chaining and per-hash-bucket locking, and Figure 10.1 diagrams how they fit together. The `hashtab` structure (lines 11–15 in Listing 10.1) contains four `ht_bucket` structures (lines 6–9 in Listing 10.1), with the `->ht_nbuckets` field controlling the number of buckets and the `->ht_cmp` field holding the pointer to key-comparison function. Each such bucket contains a list header `->htb_head` and a lock `->htb_lock`. The list headers chain `ht_elem` structures (lines 1–4 in Listing 10.1) through their `->hte_next` fields, and each `ht_elem` structure also caches the corresponding element's hash value in the `->hte_hash` field. The `ht_elem` structure is included in a larger structure which might contain a complex key.

Figure 10.1 shows bucket 0 containing two elements and bucket 2 containing one.

Listing 10.2 shows mapping and locking functions. Lines 1 and 2 show the macro `HASH2BKT()`, which maps from a hash value to the corresponding `ht_bucket` structure. This macro uses a simple modulus: if more aggressive hashing is required, the caller needs to implement it when mapping from key to hash value. The remaining two functions acquire and release the `->htb_lock` corresponding to the specified hash value.

Listing 10.1: Hash-Table Data Structures

```
1 struct ht_elem {
2   struct cds_list_head hte_next;
3   unsigned long hte_hash;
4 };
5
6 struct ht_bucket {
7   struct cds_list_head htb_head;
8   spinlock_t htb_lock;
9 };
10
11 struct hashtab {
12   unsigned long ht_nbuckets;
13   int (*ht_cmp)(struct ht_elem *htep, void *key);
14   struct ht_bucket ht_bkt[0];
15 };
```

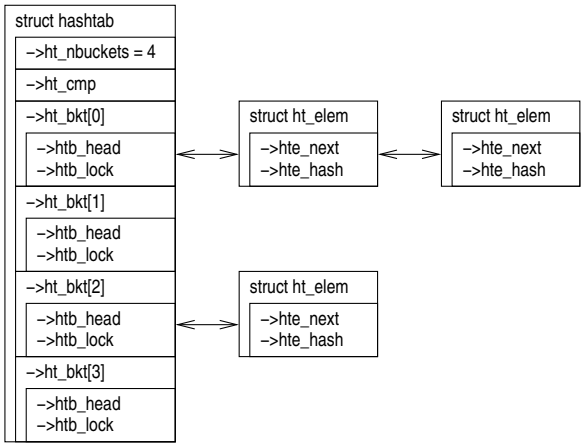


Figure 10.1: Hash-Table Data-Structure Diagram

Listing 10.3 shows `hashtab_lookup()`, which returns a pointer to the element with the specified hash and key if it exists, or `NULL` otherwise. This function takes both a hash value and a pointer to the key because this allows users of this function to use arbitrary keys and arbitrary hash functions. Line 8 maps from the hash value to a pointer to the corresponding hash bucket. Each pass through the loop spanning lines 9–14 examines one element of the bucket's hash chain. Line 10 checks to see if the hash values match, and if not, line 11 proceeds to the next element. Line 12 checks to see if the actual key matches, and if so, line 13 returns a pointer to the matching element. If no element matches, line 15 returns `NULL`.

Quick Quiz 10.2: But isn't the double comparison on lines 10–13 in Listing 10.3 inefficient in the case where the key fits into an unsigned long? ■

Listing 10.2: Hash-Table Mapping and Locking

```
1 #define HASH2BKT(htp, h) \
2   (&(htp)->ht_bkt[h % (htp)->ht_nbuckets])
3
4 static void hashtab_lock(struct hashtab *htp,
5                           unsigned long hash)
6 {
7   spin_lock(&HASH2BKT(htp, hash)->htb_lock);
8 }
9
10 static void hashtab_unlock(struct hashtab *htp,
11                             unsigned long hash)
12 {
13   spin_unlock(&HASH2BKT(htp, hash)->htb_lock);
14 }
```

Listing 10.3: Hash-Table Lookup

```
1 struct ht_elem *
2 hashtab_lookup(struct hashtab *htp, unsigned long hash,
3                void *key)
4 {
5   struct ht_bucket *htb;
6   struct ht_elem   *htep;
7
8   htb = HASH2BKT(htp, hash);
9   cds_list_for_each_entry(htep, &htb->htb_head, hte_next) {
10    if (htep->hte_hash != hash)
11      continue;
12    if (htp->ht_cmp(htep, key))
13      return htep;
14  }
15  return NULL;
16 }
```

Listing 10.4: Hash-Table Modification

```
1 void hashtab_add(struct hashtab *htp, unsigned long hash,
2                  struct ht_elem *htep)
3 {
4   htep->hte_hash = hash;
5   cds_list_add(&htep->hte_next,
6                &HASH2BKT(htp, hash)->htb_head);
7 }
8
9 void hashtab_del(struct ht_elem *htep)
10 {
11   cds_list_del_init(&htep->hte_next);
12 }
```

Listing 10.5: Hash-Table Allocation and Free

```
1 struct hashtab *
2 hashtab_alloc(unsigned long nbuckets,
3               int (*cmp)(struct ht_elem *htep, void *key))
4 {
5   struct hashtab *htp;
6   int i;
7
8   htp = malloc(sizeof(*htp) +
9                nbuckets * sizeof(struct ht_bucket));
10  if (htp == NULL)
11    return NULL;
12  htp->ht_nbuckets = nbuckets;
13  htp->ht_cmp = cmp;
14  for (i = 0; i < nbuckets; i++) {
15    CDS_INIT_LIST_HEAD(&htp->ht_bkt[i].htb_head);
16    spin_lock_init(&htp->ht_bkt[i].htb_lock);
17  }
18  return htp;
19 }
20
21 void hashtab_free(struct hashtab *htp)
22 {
23   free(htp);
24 }
```

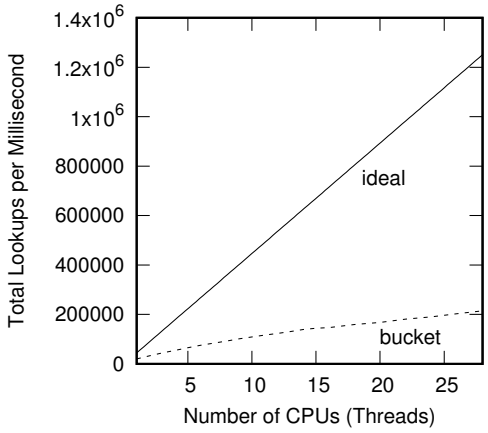


Figure 10.2: Read-Only Hash-Table Performance For Schrödinger's Zoo

Listing 10.4 shows the `hashtab_add()` and `hashtab_del()` functions that add and delete elements from the hash table, respectively.

The `hashtab_add()` function simply sets the element's hash value on line 4, then adds it to the corresponding bucket on lines 5 and 6. The `hashtab_del()` function simply removes the specified element from whatever hash chain it is on, courtesy of the doubly linked nature of the hash-chain lists. Before calling either of these two functions, the caller is required to ensure that no other thread is accessing or modifying this same bucket, for example, by invoking `hashtab_lock()` beforehand.

Listing 10.5 shows `hashtab_alloc()` and `hashtab_free()`, which do hash-table allocation and freeing, respectively. Allocation begins on lines 8–9 with allocation of the underlying memory. If line 10 detects that memory has been exhausted, line 11 returns `NULL` to the caller. Otherwise, lines 12 and 13 initialize the number of buckets and the pointer to key-comparison function, and the loop spanning lines 14–17 initializes the buckets themselves, including the chain list header on line 15 and the lock on line 16. Finally, line 18 returns a pointer to the newly allocated hash table. The `hashtab_free()` function on lines 21–24 is straightforward.

www.dbooks.org

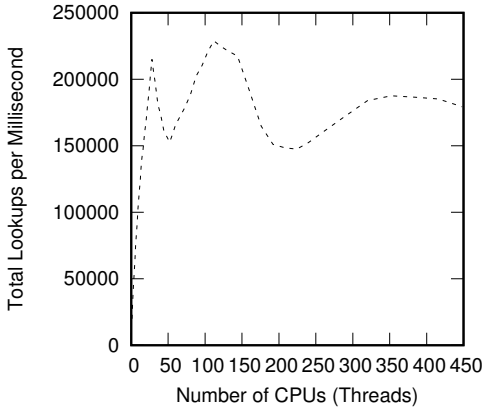


Figure 10.3: Read-Only Hash-Table Performance For Schrödinger's Zoo, 448 CPUs

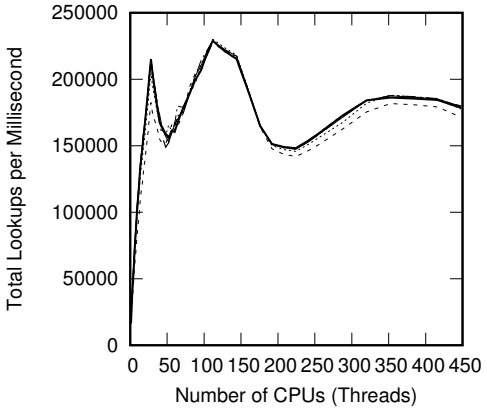


Figure 10.4: Read-Only Hash-Table Performance For Schrödinger's Zoo, Varying Buckets

10.2.3 Hash-Table Performance

The performance results for a single 28-core socket of a 2.1 GHz Intel Xeon system using a bucket-locked hash table with 262,144 buckets are shown in Figure 10.2. The performance does scale nearly linearly, but it falls a far short of the ideal performance level, even at only 28 CPUs. Part of this shortfall is due to the fact that the lock acquisitions and releases incur no cache misses on a single CPU, but do incur misses on two or more CPUs.

And things only get worse with more CPUs, as can be seen in Figure 10.3. We do not need to show ideal performance: The performance for 29 CPUs and beyond is all too clearly worse than abysmal. This clearly underscores the dangers of extrapolating performance from a modest number of CPUs.

Of course, one possible reason for the collapse in performance might be that more hash buckets are needed. We can test this by increasing the number of hash buckets.

Quick Quiz 10.3: Instead of simply increasing the number of hash buckets, wouldn't it be better to cache-align the existing hash buckets? ■

However, as can be seen in Figure 10.4, changing the number of buckets has almost no effect: Scalability is still abysmal. In particular, we still see a sharp dropoff at 29 CPUs and beyond. Clearly something else is going on.

The problem is that this is a multi-socket system, with CPUs 0–27 and 225–251 mapped to the first socket as shown in Figure 10.5. Test runs confined to the first 28 CPUs therefore perform quite well, but tests that involve socket 0's CPUs 0–27 as well as socket 1's CPU 28 incur the overhead of passing data across socket boundaries. This can severely degrade performance, as was discussed in Section 3.2.1. In short, large multi-socket systems require good locality of reference in addition to full partitioning. The remainder of this chapter will discuss ways of providing good locality of reference within the hash table itself, but in the meantime please note that one other way to provide good locality of reference would be to place large data elements in the hash table. For example, Schrödinger might attain excellent cache locality by placing photographs or even videos of his animals in each element of the hash table. But for those needing hash tables containing small data elements, please read on!

Quick Quiz 10.4: Given the negative scalability of the Schrödinger's Zoo application across sockets, why not just run multiple copies of the application, with each copy having a subset of the animals and confined to run on a single socket? ■

One key property of the Schrödinger's-zoo runs discussed thus far is that they are all read-only. This makes the performance degradation due to lock-acquisition-induced cache misses all the more painful. Even though we are not updating the underlying hash table itself, we are still paying the price for writing to memory. Of course, if the hash table was never going to be updated, we could dispense entirely with mutual exclusion. This approach is quite straightforward and is left as an exercise for the reader. But even with the occasional update, avoiding writes avoids cache misses, and allows the read-mostly data to be replicated across all the caches, which in turn promotes locality of reference.

Socket	Hyperthread 0	Hyperthread 1
0	0–27	224–251
1	28–55	252–279
2	56–83	280–307
3	84–111	308–335
4	112–139	336–363
5	140–167	364–391
6	168–195	392–419
7	196–223	420–447

Figure 10.5: NUMA Topology of System Under Test

The next section therefore examines optimizations that can be carried out in read-mostly cases where updates are rare, but could happen at any time.

10.3 Read-Mostly Data Structures

> Adapt the remedy to the disease.
>
> *Chinese proverb*

Although partitioned data structures can offer excellent scalability, NUMA effects can result in severe degradations of both performance and scalability. In addition, the need for read-side synchronization can degrade performance in read-mostly situations. However, we can achieve both performance and scalability by using RCU, which was introduced in Section 9.5. Similar results can be achieved using hazard pointers (`hazptr.c`) [Mic04], which will be included in the performance results shown in this section [McK13].

10.3.1 RCU-Protected Hash Table Implementation

For an RCU-protected hash table with per-bucket locking, updaters use locking as shown in Section 10.2, but readers use RCU. The data structures remain as shown in Listing 10.1, and the `HASH2BKT()`, `hashtab_lock()`, and `hashtab_unlock()` functions remain as shown in Listing 10.2. However, readers use the lighter-weight concurrency-control embodied by `hashtab_lock_lookup()` and `hashtab_unlock_lookup()` shown in Listing 10.6.

Listing 10.6: RCU-Protected Hash-Table Read-Side Concurrency Control

```
1 static void hashtab_lock_lookup(struct hashtab *htp,
2                                 unsigned long hash)
3 {
4   rcu_read_lock();
5 }
6
7 static void hashtab_unlock_lookup(struct hashtab *htp,
8                                   unsigned long hash)
9 {
10  rcu_read_unlock();
11 }
```

Listing 10.7: RCU-Protected Hash-Table Lookup

```
1  struct ht_elem *hashtab_lookup(struct hashtab *htp,
2                                 unsigned long hash,
3                                 void *key)
4  {
5    struct ht_bucket *htb;
6    struct ht_elem  *htep;
7
8    htb = HASH2BKT(htp, hash);
9    cds_list_for_each_entry_rcu(htep,
10                               &htb->htb_head,
11                               hte_next) {
12     if (htep->hte_hash != hash)
13       continue;
14     if (htp->ht_cmp(htep, key))
15       return htep;
16   }
17   return NULL;
18 }
```

Listing 10.7 shows `hashtab_lookup()` for the RCU-protected per-bucket-locked hash table. This is identical to that in Listing 10.3 except that `cds_list_for_each_entry()` is replaced by `cds_list_for_each_entry_rcu()`. Both of these primitives traverse the hash chain referenced by `htb->htb_head` but `cds_list_for_each_entry_rcu()` also correctly enforces memory ordering in case of concurrent insertion. This is an important difference between these two hash-table implementations: Unlike the pure per-bucket-locked implementation, the RCU protected implementation allows lookups to run concurrently with insertions and deletions, and RCU-aware primitives like `cds_list_for_each_entry_rcu()` are required to correctly handle this added concurrency. Note also that `hashtab_lookup()`'s caller must be within an RCU read-side critical section, for example, the caller must invoke `hashtab_lock_lookup()` before invoking `hashtab_lookup()` (and of course invoke `hashtab_unlock_lookup()` some time afterwards).

Quick Quiz 10.5: But if elements in a hash table can be removed concurrently with lookups, doesn't that mean that a lookup could return a reference to a data element that was removed immediately after it was looked up? ■

www.dbooks.org

Listing 10.8: RCU-Protected Hash-Table Modification

```
1 void hashtab_add(struct hashtab *htp,
2                   unsigned long hash,
3                   struct ht_elem *htep)
4 {
5   htep->hte_hash = hash;
6   cds_list_add_rcu(&htep->hte_next,
7                    &HASH2BKT(htp, hash)->htb_head);
8 }
9
10 void hashtab_del(struct ht_elem *htep)
11 {
12   cds_list_del_rcu(&htep->hte_next);
13 }
```

Listing 10.8 shows `hashtab_add()` and `hashtab_del()`, both of which are quite similar to their counterparts in the non-RCU hash table shown in Listing 10.4. The `hashtab_add()` function uses `cds_list_add_rcu()` instead of `cds_list_add()` in order to ensure proper ordering when an element is added to the hash table at the same time that it is being looked up. The `hashtab_del()` function uses `cds_list_del_rcu()` instead of `cds_list_del_init()` to allow for the case where an element is looked up just before it is deleted. Unlike `cds_list_del_init()`, `cds_list_del_rcu()` leaves the forward pointer intact, so that `hashtab_lookup()` can traverse to the newly deleted element's successor.

Of course, after invoking `hashtab_del()`, the caller must wait for an RCU grace period (e.g., by invoking `synchronize_rcu()`) before freeing or otherwise reusing the memory for the newly deleted element.

10.3.2 RCU-Protected Hash Table Performance

Figure 10.6 shows the read-only performance of RCU-protected and hazard-pointer-protected hash tables against the previous section's per-bucket-locked implementation. As you can see, both RCU and hazard pointers perform and scale much better than per-bucket locking because read-only replication avoids NUMA effects. The difference increases with larger numbers of threads. Results from a globally locked implementation are also shown, and as expected the results are even worse than those of the per-bucket-locked implementation. RCU does slightly better than hazard pointers.

Figure 10.7 shows the same data on a linear scale. This drops the global-locking trace into the x-axis, but allows the non-ideal performance of RCU and hazard pointers to be more readily discerned. Both show a change in slope at 224 CPUs, and this is due to hardware multithreading. At 32 and fewer CPUs, each thread has a core to itself. In this regime, RCU does better than does hazard pointers because the latter's read-side memory barriers result in dead time within the core. In short, RCU is better able to utilize a core from a single hardware thread than is hazard pointers.

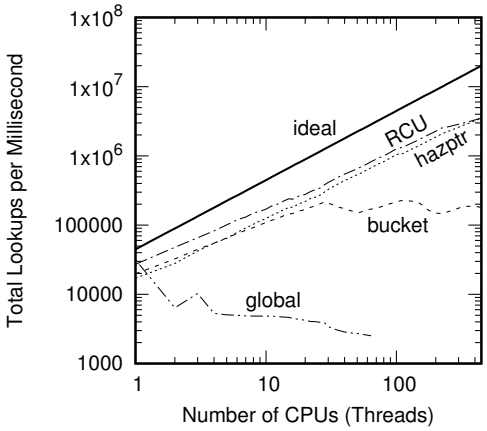


Figure 10.6: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo

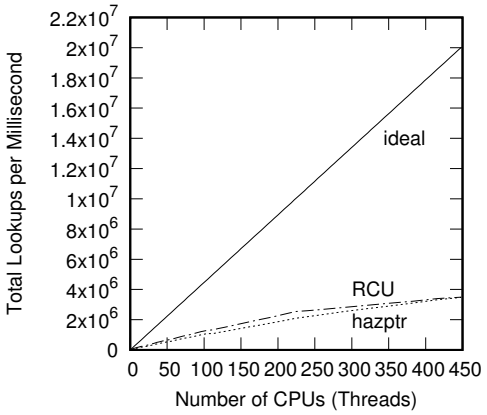


Figure 10.7: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo, Linear Scale

This situation changes above 224 CPUs. Because RCU is using more than half of each core's resources from a single hardware thread, RCU gains relatively little benefit from the second hardware thread in each core. The slope of the hazard-pointers trace also decreases at 224 CPUs, but less dramatically, because the second hardware thread is able to fill in the time that the first hardware thread is stalled due to memory-barrier latency. As we will see in later sections, this second-hardware-thread advantage depends on the workload.

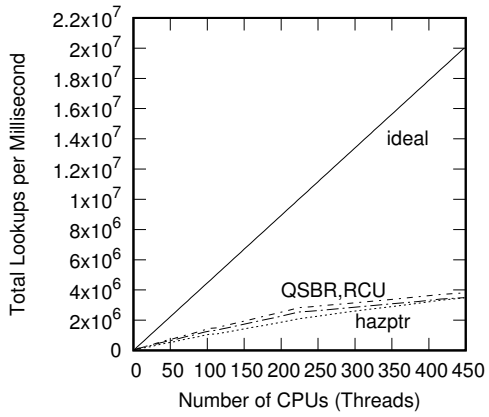


Figure 10.8: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo including QSBR, Linear Scale

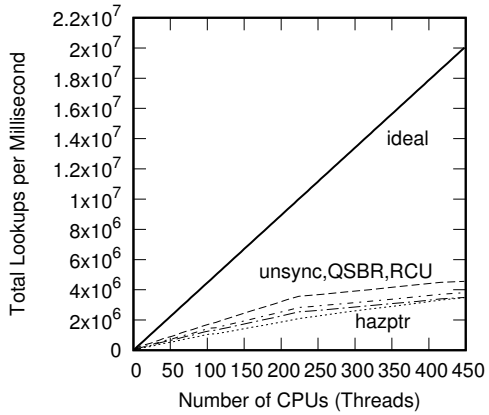


Figure 10.9: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo including QSBR and Unsynchronized, Linear Scale

But why is RCU's performance a factor of five less than ideal? One possibility is that the per-thread counters manipulated by `rcu_read_lock()` and `rcu_read_unlock()` are slowing things down. Figure 10.8 therefore adds the results for the QSBR variant of RCU, whose read-side primitives do nothing. And although QSBR does perform slightly better than does RCU, it is still about a factor of five short of ideal.

Figure 10.9 adds completely unsynchronized results, which works because this is a read-only benchmark with nothing to synchronize. Even with no synchronization whatsoever, performance still falls far short of ideal.

The problem is that this system has sockets with 28 cores, which have the modest cache sizes shown in Figure 3.2 on page 24. Each hash bucket (`struct ht_bucket`) occupies 56 bytes and each element (`struct zoo_he`) occupies 72 bytes for the RCU and QSBR runs. The benchmark generating Figure 10.9 used 262,144 buckets and up to 262,144 elements, for a total of 33,554,448 bytes, which not only overflows the 1,048,576-byte L2 caches by more than a factor of thirty, but is also uncomfortably close to the L3 cache size of 40,370,176 bytes, especially given that this cache has only 11 ways. This means that L2 cache collisions will be the rule and also that L3 cache collisions will not be uncommon, so that the resulting cache misses will degrade performance. In this case, the bottleneck is not in the CPU, but rather in the hardware memory system.

Additional evidence for this memory-system bottleneck may be found by examining the unsynchronized code. This code does not need locks, so each hash bucket occupies only 16 bytes compared to the 56 bytes for RCU and QSBR. signal-based). Similarly, each hash-table element occupies only 56 bytes compared to the 72 bytes for RCU and QSBR. So it is unsurprising that the single-CPU unsynchronized run performs up to about half again faster than that of either QSBR or RCU.

Quick Quiz 10.6: How can we be so sure that the hash-table size is at fault here, especially given that Figure 10.4 on page 178 shows that varying hash-table size has almost no effect? Might the problem instead be something like false sharing? ■

What if the memory footprint is reduced still further? Figure 9.22 on page 485 shows that RCU attains very nearly ideal performance on the much smaller data structure represented by the pre-BSD routing table.

Quick Quiz 10.7: The memory system is a serious bottleneck on this big system. Why bother putting 448 CPUs on a system without giving them enough memory bandwidth to do something useful??? ■

As noted earlier, Schrödinger is surprised by the popularity of his cat [Sch35], but recognizes the need to reflect this popularity in his design. Figure 10.10 shows the results of 64-CPU runs, varying the number of CPUs that are doing nothing but looking up the cat. Both RCU and hazard pointers respond well to this challenge, but bucket locking scales negatively, eventually performing as badly as global locking. This should not be a surprise because if all CPUs are doing nothing but looking up the cat, the lock corresponding to the cat's bucket is for all intents and purposes a global lock.

www.dbooks.org

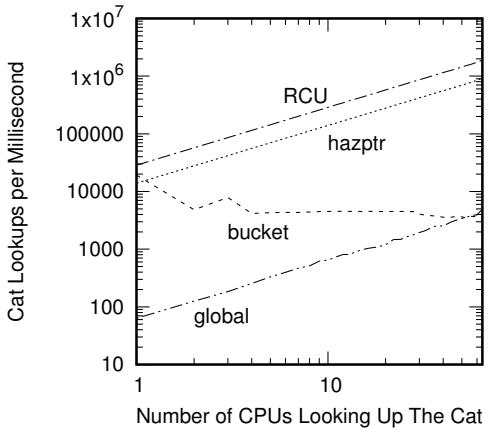


Figure 10.10: Read-Side Cat-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo at 64 CPUs

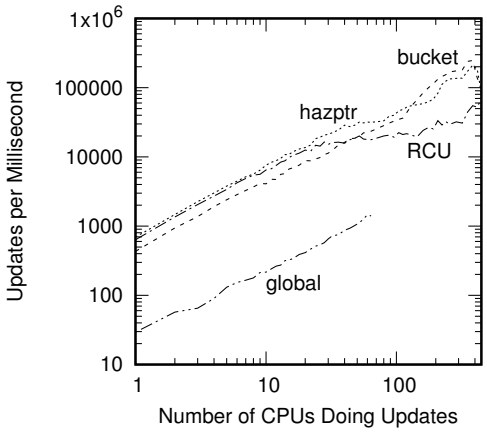


Figure 10.12: Update-Side RCU-Protected Hash-Table Performance For Schrödinger's Zoo

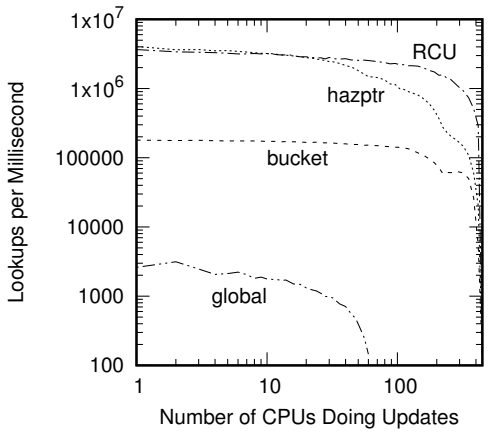


Figure 10.11: Read-Side RCU-Protected Hash-Table Performance For Schrödinger's Zoo in the Presence of Updates

This cat-only benchmark illustrates one potential problem with fully partitioned sharding approaches. Only the CPUs associated with the cat's partition is able to access the cat, limiting the cat-only throughput. Of course, a great many applications have good load-spreading properties, and for these applications sharding works quite well. However, sharding does not handle "hot spots" very well, with the hot spot exemplified by Schrödinger's cat being but one case in point.

If we were only ever going to read the data, we would not need any concurrency control to begin with. Figure 10.11 therefore shows the effect of updates on readers. At the extreme left-hand side of this graph, all but one of the CPUs are doing lookups, while to the right all 448 CPUs are doing updates. For all four implementations, the number of lookups per millisecond decreases as the number of updating CPUs increases, of course reaching zero lookups per millisecond when all 448 CPUs are updating. Both hazard pointers and RCU do well compared to per-bucket locking because their readers do not increase update-side lock contention. RCU does well relative to hazard pointers as the number of updaters increases due to the latter's read-side memory barriers, which incur greater overhead, especially in the presence of updates, and particularly when execution involves more than one socket. It therefore seems likely that modern hardware heavily optimizes memory-barrier execution, greatly reducing memory-barrier overhead in the read-only case.

Where Figure 10.11 showed the effect of increasing update rates on lookups, Figure 10.12 shows the effect of increasing update rates on the updates themselves. Again, at the left-hand side of the figure all but one of the CPUs are doing lookups and at the right-hand side of the figure all 448 CPUs are doing updates. Hazard pointers and RCU start off with a significant advantage because, unlike bucket locking, readers do not exclude updaters. However, as the number of updating CPUs increases, update-side overhead starts to make its presence known, first for RCU and then for hazard pointers. Of course, all three of these implementations beat global locking.

It is quite possible that the differences in lookup performance are affected by the differences in update rates. One way to check this is to artificially throttle the update rates of per-bucket locking and hazard pointers to match that of RCU. Doing so does not significantly improve the lookup performance of per-bucket locking, nor does it

close the gap between hazard pointers and RCU. However, removing the read-side memory barriers from hazard pointers (thus resulting in an unsafe implementation) does nearly close the gap between hazard pointers and RCU. Although this unsafe hazard-pointer implementation will usually be reliable enough for benchmarking purposes, it is absolutely not recommended for production use.

Quick Quiz 10.8: The dangers of extrapolating from 28 CPUs to 448 CPUs was made quite clear in Section 10.2.3. But why should extrapolating up from 448 CPUs be any safer? ■

10.3.3 RCU-Protected Hash Table Discussion

One consequence of the RCU and hazard-pointer implementations is that a pair of concurrent readers might disagree on the state of the cat. For example, one of the readers might have fetched the pointer to the cat's data structure just before it was removed, while another reader might have fetched this same pointer just afterwards. The first reader would then believe that the cat was alive, while the second reader would believe that the cat was dead.

This situation is completely fitting for Schrödinger's cat, but it turns out that it is quite reasonable for normal non-quantum cats as well. After all, it is impossible to determine exactly when an animal is born or dies.

To see this, let's suppose that we detect a cat's death by heartbeat. This raise the question of exactly how long we should wait after the last heartbeat before declaring death. It is clearly ridiculous to wait only one millisecond, because then a healthy living cat would have to be declared dead—and then resurrected—more than once per second. It is equally ridiculous to wait a full month, because by that time the poor cat's death would have made itself very clearly known via olfactory means.

Because an animal's heart can stop for some seconds and then start up again, there is a tradeoff between timely recognition of death and probability of false alarms. It is quite possible that a pair of veterinarians might disagree on the time to wait between the last heartbeat and the declaration of death. For example, one veterinarian might declare death thirty seconds after the last heartbeat, while another might insist on waiting a full minute. In this case, the two veterinarians would disagree on the state of the cat for the second period of thirty seconds following the last heartbeat, as fancifully depicted in Figure 10.13.

Heisenberg taught us to live with this sort of uncertainty [Hei27], which is a good thing because computing



Figure 10.13: Even Veterinarians Disagree!

hardware and software acts similarly. For example, how do you know that a piece of computing hardware has failed? Often because it does not respond in a timely fashion. Just like the cat's heartbeat, this results in a window of uncertainty as to whether or not the hardware has really failed, as opposed to just being slow.

Furthermore, most computing systems are intended to interact with the outside world. Consistency with the outside world is therefore of paramount importance. However, as we saw in Figure 9.26 on page 159, increased internal consistency can come at the expense of degraded external consistency. Techniques such as RCU and hazard pointers give up some degree of internal consistency to attain improved external consistency.

In short, internal consistency is not necessarily a natural part of all problem domains, and often incurs great expense in terms of performance, scalability, consistency with the outside world [HKLP12, HHK+13, Rin13], or all of the above.

10.4 Non-Partitionable Data Structures

> Don't be afraid to take a big step if one is indicated. You can't cross a chasm in two small steps.
>
> *David Lloyd GeorgDavid Lloyd George*

Fixed-size hash tables are perfectly partitionable, but resizable hash tables pose partitioning challenges when growing or shrinking, as fancifully depicted in Figure 10.14. However, it turns out that it is possible to construct high-

www.dbooks.org

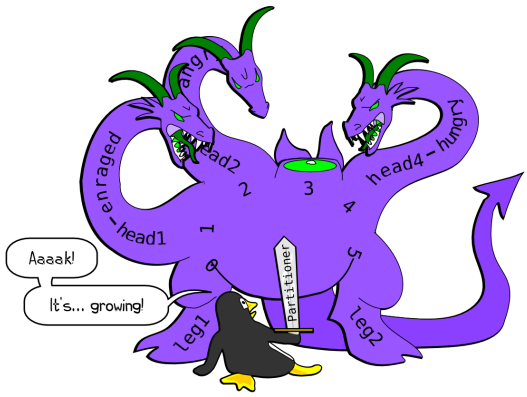


Figure 10.14: Partitioning Problems

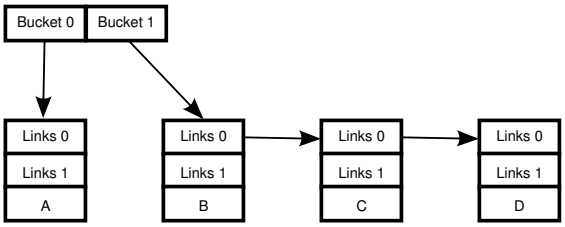


Figure 10.15: Growing a Two-List Hash Table, State (a)

performance scalable RCU-protected hash tables, as described in the following sections.

10.4.1 Resizable Hash Table Design

In happy contrast to the situation in the early 2000s, there are now no fewer than three different types of scalable RCU-protected hash tables. The first (and simplest) was developed for the Linux kernel by Herbert Xu [Xu10], and is described in the following sections. The other two are covered briefly in Section 10.4.4.

The key insight behind the first hash-table implementation is that each data element can have two sets of list pointers, with one set currently being used by RCU readers (as well as by non-RCU updaters) and the other being used to construct a new resized hash table. This approach allows lookups, insertions, and deletions to all run concurrently with a resize operation (as well as with each other).

The resize operation proceeds as shown in Figures 10.15–10.18, with the initial two-bucket state shown in Figure 10.15 and with time advancing from figure to figure. The initial state uses the zero-index links to chain

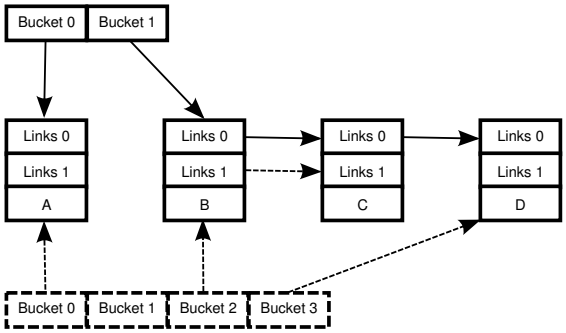


Figure 10.16: Growing a Two-List Hash Table, State (b)

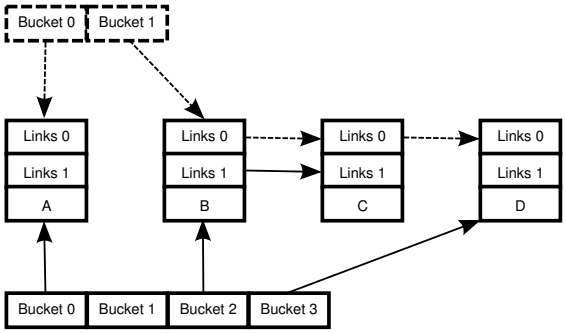


Figure 10.17: Growing a Two-List Hash Table, State (c)

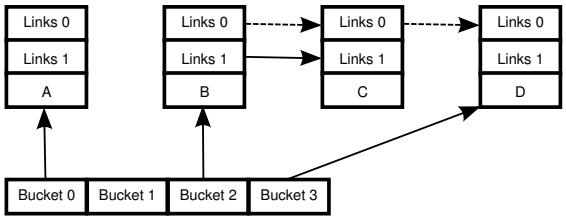


Figure 10.18: Growing a Two-List Hash Table, State (d)

the elements into hash buckets. A four-bucket array is allocated, and the one-index links are used to chain the elements into these four new hash buckets. This results in state (b) shown in Figure 10.16, with readers still using the original two-bucket array.

The new four-bucket array is exposed to readers and then a grace-period operation waits for all readers, resulting in state (c), shown in Figure 10.17. In this state, all readers are using the new four-bucket array, which means that the old two-bucket array may now be freed, resulting in state (d), shown in Figure 10.18.

Listing 10.9: Resizable Hash-Table Data Structures

```
1 struct ht_elem {
2   struct rcu_head rh;
3   struct cds_list_head hte_next[2];
4 };
5
6 struct ht_bucket {
7   struct cds_list_head htb_head;
8   spinlock_t htb_lock;
9 };
10
11 struct ht {
12   long ht_nbuckets;
13   long ht_resize_cur;
14   struct ht *ht_new;
15   int ht_idx;
16   int (*ht_cmp)(struct ht_elem *htep, void *key);
17   unsigned long (*ht_gethash)(void *key);
18   void *(*ht_getkey)(struct ht_elem *htep);
19   struct ht_bucket ht_bkt[0];
20 };
21
22 struct ht_lock_state {
23   struct ht_bucket *hbp[2];
24   int hls_idx[2];
25 };
26
27 struct hashtab {
28   struct ht *ht_cur;
29   spinlock_t ht_lock;
30 };
```

This design leads to a relatively straightforward implementation, which is the subject of the next section.

10.4.2 Resizable Hash Table Implementation

Resizing is accomplished by the classic approach of inserting a level of indirection, in this case, the `ht` structure shown on lines 11–20 of Listing 10.9 (`hash_resize.c`). The `hashtab` structure shown on lines 27–30 contains only a pointer to the current `ht` structure along with a spinlock that is used to serialize concurrent attempts to resize the hash table. If we were to use a traditional lock- or atomic-operation-based implementation, this `hashtab` structure could become a severe bottleneck from both performance and scalability viewpoints. However, because resize operations should be relatively infrequent, we should be able to make good use of RCU.

The `ht` structure represents a specific size of the hash table, as specified by the `->ht_nbuckets` field on line 12. The size is stored in the same structure containing the array of buckets (`->ht_bkt[]` on line 19) in order to avoid mismatches between the size and the array. The `->ht_resize_cur` field on line 13 is equal to −1 unless a resize operation is in progress, in which case it indicates the index of the bucket whose elements are being inserted into the new hash table, which is referenced by the `->ht_new` field on line 14. If there is no resize operation in progress, `->ht_new` is `NULL`. Thus, a resize operation proceeds by allocating a new `ht` structure and referencing it via the `->ht_new` pointer, then advancing `->ht_resize_cur` through the old table's buckets. When all the elements have been added to the new table, the new table is linked into the `hashtab` structure's `->ht_cur` field. Once all old readers have completed, the old hash table's `ht` structure may be freed.

The `->ht_idx` field on line 15 indicates which of the two sets of list pointers are being used by this instantiation of the hash table, and is used to index the `->hte_next[]` array in the `ht_elem` structure on line 3.

The `->ht_cmp()`, `->ht_gethash()`, and `->ht_getkey()` fields on lines 16–18 collectively define the per-element key and the hash function. The `->ht_cmp()` function compares a specified key with that of the specified element, the `->ht_gethash()` calculates the specified key's hash, and `->ht_getkey()` extracts the key from the enclosing data element.

The `ht_lock_state` shown on lines 22–25 is used to communicate lock state from a new `hashtab_lock_mod()` to `hashtab_add()`, `hashtab_del()`, and `hashtab_unlock_mod()`. This state prevents the algorithm from being redirected to the wrong bucket during concurrent resize operations.

The `ht_bucket` structure is the same as before, and the `ht_elem` structure differs from that of previous implementations only in providing a two-element array of list pointer sets in place of the prior single set of list pointers.

In a fixed-sized hash table, bucket selection is quite straightforward: Simply transform the hash value to the corresponding bucket index. In contrast, when resizing, it is also necessary to determine which of the old and new sets of buckets to select from. If the bucket that would be selected from the old table has already been distributed into the new table, then the bucket should be selected from the new table as well as from the old table. Conversely, if the bucket that would be selected from the old table has not yet been distributed, then the bucket should be selected from the old table.

Bucket selection is shown in Listing 10.10, which shows `ht_get_bucket()` on lines 1–11 and `ht_search_bucket()` on lines 13–28. The `ht_get_bucket()` function returns a reference to the bucket corresponding to the specified key in the specified hash table, without making any allowances for resizing. It also stores the bucket index corresponding to the key into the location referenced by

www.dbooks.org

Listing 10.10: Resizable Hash-Table Bucket Selection

```
1 static struct ht_bucket *
2 ht_get_bucket(struct ht *htp, void *key,
3               long *b, unsigned long *h)
4 {
5   unsigned long hash = htp->ht_gethash(key);
6
7   *b = hash % htp->ht_nbuckets;
8   if (h)
9     *h = hash;
10  return &htp->ht_bkt[*b];
11 }
12
13 static struct ht_elem *
14 ht_search_bucket(struct ht *htp, void *key)
15 {
16   long b;
17   struct ht_elem *htep;
18   struct ht_bucket *htbp;
19
20   htbp = ht_get_bucket(htp, key, &b, NULL);
21   cds_list_for_each_entry_rcu(htep,
22                               &htbp->htb_head,
23                               hte_next[htp->ht_idx]) {
24     if (htp->ht_cmp(htep, key))
25       return htep;
26   }
27   return NULL;
28 }
```

Listing 10.11: Resizable Hash-Table Update-Side Concurrency Control

```
1 static void
2 hashtab_lock_mod(struct hashtab *htp_master, void *key,
3                  struct ht_lock_state *lsp)
4 {
5   long b;
6   unsigned long h;
7   struct ht *htp;
8   struct ht_bucket *htbp;
9
10  rcu_read_lock();
11  htp = rcu_dereference(htp_master->ht_cur);
12  htbp = ht_get_bucket(htp, key, &b, &h);
13  spin_lock(&htbp->htb_lock);
14  lsp->hbp[0] = htbp;
15  lsp->hls_idx[0] = htp->ht_idx;
16  if (b > READ_ONCE(htp->ht_resize_cur)) {
17    lsp->hbp[1] = NULL;
18    return;
19  }
20  htp = rcu_dereference(htp->ht_new);
21  htbp = ht_get_bucket(htp, key, &b, &h);
22  spin_lock(&htbp->htb_lock);
23  lsp->hbp[1] = htbp;
24  lsp->hls_idx[1] = htp->ht_idx;
25 }
26
27 static void
28 hashtab_unlock_mod(struct ht_lock_state *lsp)
29 {
30  spin_unlock(&lsp->hbp[0]->htb_lock);
31  if (lsp->hbp[1])
32    spin_unlock(&lsp->hbp[1]->htb_lock);
33  rcu_read_unlock();
34 }
```

parameter b on line 7, and the corresponding hash value corresponding to the key into the location referenced by parameter h (if non-NULL) on line 9. Line 10 then returns a reference to the corresponding bucket.

The ht_search_bucket() function searches for the specified key within the specified hash-table version. Line 20 obtains a reference to the bucket corresponding to the specified key. The loop spanning lines 21–26 searches that bucket, so that if line 24 detects a match, line 25 returns a pointer to the enclosing data element. Otherwise, if there is no match, line 27 returns NULL to indicate failure.

Quick Quiz 10.9: How does the code in Listing 10.10 protect against the resizing process progressing past the selected bucket? ■

This implementation of ht_get_bucket() and ht_search_bucket() permits lookups and modifications to run concurrently with a resize operation.

Read-side concurrency control is provided by RCU as was shown in Listing 10.6, but the update-side concurrency-control functions hashtab_lock_mod() and hashtab_unlock_mod() must now deal with the possibility of a concurrent resize operation as shown in Listing 10.11.

The hashtab_lock_mod() spans lines 1–25 in the listing. Line 10 enters an RCU read-side critical section to prevent the data structures from being freed during the traversal, line 11 acquires a reference to the current hash table, and then line 12 obtains a reference to the bucket in this hash table corresponding to the key. Line 13 acquires that bucket's lock, which will prevent any concurrent resizing operation from distributing that bucket, though of course it will have no effect if that bucket has already been distributed. Lines 14–15 store the bucket pointer and pointer-set index into their respective fields in the ht_lock_state structure, which communicates the information to hashtab_add(), hashtab_del(), and hashtab_unlock_mod(). Line 16 then checks to see if a concurrent resize operation has already distributed this bucket across the new hash table, and if not, line 17 indicates that there is no already-resized hash bucket and line 18 returns with the selected hash bucket's lock held (thus preventing a concurrent resize operation from distributing this bucket) and also within an RCU read-side critical section. Deadlock is avoided because the old table's locks are always acquired before those of the new table, and because the use of RCU prevents more than two versions from existing at a given time, thus preventing a deadlock cycle.

Otherwise, a concurrent resize operation has already distributed this bucket, so line 20 proceeds to the new hash table, line 21 selects the bucket corresponding to the key, and line 22 acquires the bucket's lock. Lines 23–24 store the bucket pointer and pointer-set index into their respective fields in the `ht_lock_state` structure, which again communicates this information to `hashtab_add()`, `hashtab_del()`, and `hashtab_unlock_mod()`. Because this bucket has already been resized and because `hashtab_add()` and `hashtab_del()` affect both the old and the new `ht_bucket` structures, two locks are held, one on each of the two buckets. Additionally, both elements of each array in `ht_lock_state` structure are used, with the `[0]` element pertaining to the old `ht_bucket` structure and the `[1]` element pertaining to the new structure. Once again, `hashtab_lock_mod()` exits within an RCU read-side critical section.

The `hashtab_unlock_mod()` function releases the lock(s) acquired by `hashtab_lock_mod()`. Line 30 releases the lock on the old `ht_bucket` structure. In the unlikely event that line 31 determines that a resize operation is in progress, line 32 releases the lock on the new `ht_bucket` structure. Either way, line 33 exits the RCU read-side critical section.

Quick Quiz 10.10: Suppose that one thread is inserting an element into the hash table during a resize operation. What prevents this insertion from being lost due to a subsequent resize operation completing before the insertion does? ■

Now that we have bucket selection and concurrency control in place, we are ready to search and update our resizable hash table. The `hashtab_lookup()`, `hashtab_add()`, and `hashtab_del()` functions are shown in Listing 10.12.

The `hashtab_lookup()` function on lines 1–10 of the listing does hash lookups. Line 7 fetches the current hash table and line 8 searches the bucket corresponding to the specified key. Line 9 returns a pointer to the searched-for element or `NULL` when the search fails. The caller must be within an RCU read-side critical section.

Quick Quiz 10.11: The `hashtab_lookup()` function in Listing 10.12 ignores concurrent resize operations. Doesn't this mean that readers might miss an element that was previously added during a resize operation? ■

The `hashtab_add()` function on lines 12–22 of the listing adds new data elements to the hash table. Line 15 picks up the current `ht_bucket` structure into which the new element is to be added, and line 16 picks up the index of the pointer pair. Line 18 adds the new element to the current hash bucket. If line 19 determines that this bucket has been distributed to a new version of the hash table, then line 20 also adds the new element to the corresponding new bucket. The caller is required to handle concurrency, for example, by invoking `hashtab_lock_mod()` before the call to `hashtab_add()` and invoking `hashtab_unlock_mod()` afterwards.

Listing 10.12: Resizable Hash-Table Access Functions

```
1  struct ht_elem *
2  hashtab_lookup(struct hashtab *htp_master, void *key)
3  {
4    struct ht *htp;
5    struct ht_elem *htep;
6
7    htp = rcu_dereference(htp_master->ht_cur);
8    htep = ht_search_bucket(htp, key);
9    return htep;
10 }
11
12 void hashtab_add(struct ht_elem *htep,
13                  struct ht_lock_state *lsp)
14 {
15   struct ht_bucket *htbp = lsp->hbp[0];
16   int i = lsp->hls_idx[0];
17
18   cds_list_add_rcu(&htep->hte_next[i], &htbp->htb_head);
19   if ((htbp = lsp->hbp[1])) {
20     cds_list_add_rcu(&htep->hte_next[!i], &htbp->htb_head);
21   }
22 }
23
24 void hashtab_del(struct ht_elem *htep,
25                  struct ht_lock_state *lsp)
26 {
27   int i = lsp->hls_idx[0];
28
29   cds_list_del_rcu(&htep->hte_next[i]);
30   if (lsp->hbp[1])
31     cds_list_del_rcu(&htep->hte_next[!i]);
32 }
```

The `hashtab_del()` function on lines 24–32 of the listing removes an existing element from the hash table. Line 27 picks up the index of the pointer pair and line 29 removes the specified element from the current table. If line 30 determines that this bucket has been distributed to a new version of the hash table, then line 31 also removes the specified element from the corresponding new bucket. As with `hashtab_add()`, the caller is responsible for concurrency control and this concurrency control suffices for synchronizing with a concurrent resize operation.

Quick Quiz 10.12: The `hashtab_add()` and `hashtab_del()` functions in Listing 10.12 can update two hash buckets while a resize operation is progressing. This might cause poor performance if the frequency of resize operation is not negligible. Isn't it possible to reduce the cost of updates in such cases? ■

The actual resizing itself is carried out by `hashtab_resize`, shown in Listing 10.13 on page 188. Line 16

www.dbooks.org

Listing 10.13: Resizable Hash-Table Resizing

```
1  int hashtab_resize(struct hashtab *htp_master,
2                     unsigned long nbuckets,
3                     int (*cmp)(struct ht_elem *htep, void *key),
4                     unsigned long (*gethash)(void *key),
5                     void *(*getkey)(struct ht_elem *htep))
6  {
7        struct ht *htp;
8        struct ht *htp_new;
9        int i;
10       int idx;
11       struct ht_elem *htep;
12       struct ht_bucket *htbp;
13       struct ht_bucket *htbp_new;
14       long b;
15
16       if (!spin_trylock(&htp_master->ht_lock))
17             return -EBUSY;
18       htp = htp_master->ht_cur;
19       htp_new = ht_alloc(nbuckets,
20                          cmp ? cmp : htp->ht_cmp,
21                          gethash ? gethash : htp->ht_gethash,
22                          getkey ? getkey : htp->ht_getkey);
23       if (htp_new == NULL) {
24             spin_unlock(&htp_master->ht_lock);
25             return -ENOMEM;
26       }
27       idx = htp->ht_idx;
28       htp_new->ht_idx = !idx;
29       rcu_assign_pointer(htp->ht_new, htp_new);
30       synchronize_rcu();
31       for (i = 0; i < htp->ht_nbuckets; i++) {
32             htbp = &htp->ht_bkt[i];
33             spin_lock(&htbp->htb_lock);
34             cds_list_for_each_entry(htep, &htbp->htb_head, hte_next[idx]) {
35                   htbp_new = ht_get_bucket(htp_new, htp_new->ht_getkey(htep), &b, NULL);
36                   spin_lock(&htbp_new->htb_lock);
37                   cds_list_add_rcu(&htep->hte_next[!idx], &htbp_new->htb_head);
38                   spin_unlock(&htbp_new->htb_lock);
39             }
40             WRITE_ONCE(htp->ht_resize_cur, i);
41             spin_unlock(&htbp->htb_lock);
42       }
43       rcu_assign_pointer(htp_master->ht_cur, htp_new);
44       synchronize_rcu();
45       spin_unlock(&htp_master->ht_lock);
46       free(htp);
47       return 0;
48 }
```

conditionally acquires the top-level `->ht_lock`, and if this acquisition fails, line 17 returns `-EBUSY` to indicate that a resize is already in progress. Otherwise, line 18 picks up a reference to the current hash table, and lines 19–22 allocate a new hash table of the desired size. If a new set of hash/key functions have been specified, these are used for the new table, otherwise those of the old table are preserved. If line 23 detects memory-allocation failure, line 24 releases `->ht_lock` and line 25 returns a failure indication.

Line 27 picks up the current table's index and line 28 stores its inverse to the new hash table, thus ensuring that the two hash tables avoid overwriting each other's linked lists. Line 29 then starts the bucket-distribution process by installing a reference to the new table into the `->ht_new` field of the old table. Line 30 ensures that all readers who are not aware of the new table complete before the resize operation continues.

Each pass through the loop spanning lines 31–42 distributes the contents of one of the old hash table's buckets into the new hash table. Line 32 picks up a reference to the old table's current bucket and line 33 acquires that bucket's spinlock.

Quick Quiz 10.13: In the `hashtab_resize()` function in Listing 10.13, what guarantees that the update to `->ht_new` on line 29 will be seen as happening before the update to `->ht_resize_cur` on line 40 from the perspective of `hashtab_add()` and `hashtab_del()`? In other words, what prevents

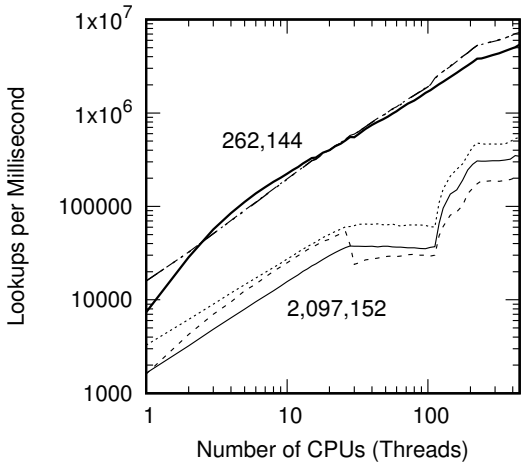


Figure 10.19: Overhead of Resizing Hash Tables Between 262,144 and 524,288 Buckets vs. Total Number of Elements

`hashtab_add()` and `hashtab_del()` from dereferencing a NULL pointer loaded from `->ht_new`? ■

Each pass through the loop spanning lines 34–39 adds one data element from the current old-table bucket to the corresponding new-table bucket, holding the new-table bucket's lock during the add operation. Line 40 updates `->ht_resize_cur` to indicate that this bucket has been distributed. Finally, line 41 releases the old-table bucket lock.

Execution reaches line 43 once all old-table buckets have been distributed across the new table. Line 43 installs the newly created table as the current one, and line 44 waits for all old readers (who might still be referencing the old table) to complete. Then line 45 releases the resize-serialization lock, line 46 frees the old hash table, and finally line 47 returns success.

Quick Quiz 10.14: Why is there a `WRITE_ONCE()` on line 40 in Listing 10.13? ■

10.4.3 Resizable Hash Table Discussion

Figure 10.19 compares resizing hash tables to their fixed-sized counterparts for 262,144 and 2,097,152 elements in the hash table. The figure shows three traces for each element count, one for a fixed-size 262,144-bucket hash table, another for a fixed-size 524,288-bucket hash table, and a third for a resizable hash table that shifts back and forth between 262,144 and 524,288 buckets, with a one-millisecond pause between each resize operation.

The uppermost three traces are for the 262,144-element hash table. The dashed trace corresponds to the two fixed-size hash tables, and the solid trace to the resizable hash table. In this case, the short hash chains cause normal lookup overhead to be so low that the overhead of resizing dominates over most of the range. In particular, the entire hash table fits into L3 cache.

The lower three traces are for the 2,097,152-element hash table. The upper trace corresponds to the 262,144-bucket fixed-size hash table, the trace in the middle for low CPU counts and at the bottom for high CPU counts to the resizable hash table, and the other trace to the 524,288-bucket fixed-size hash table. The fact that there are now an average of eight elements per bucket can only be expected to produce a sharp decrease in performance, as in fact is shown in the graph. But worse yet, the hash-table elements occupy 128 MB, which overflows each socket's 39 MB L3 cache, with performance consequences analogous to those described in Section 3.2.2. The resulting cache overflow means that the memory system is involved even for a read-only benchmark, and as you can see from the sublinear portions of the lower three traces, the memory system can be a serious bottleneck.

Quick Quiz 10.15: How much of the difference in performance between the large and small hash tables shown in Figure 10.19 was due to long hash chains and how much was due to memory-system bottlenecks? ■

Referring to the last column of Table 3.1, we recall that the first 28 CPUs are in the first socket, on a one-CPU-per-core basis, which explains the sharp decrease in performance of the resizable hash table beyond 28 CPUs. Sharp though this decrease is, please recall that it is due to constant resizing back and forth. It would clearly be better to resize once to 524,288 buckets, or, even better, do a single eight-fold resize to 2,097,152 elements, thus dropping the average number of elements per bucket down to the level enjoyed by the runs producing the upper three traces.

The key point from this data is that the RCU-protected resizable hash table performs and scales almost as well as does its fixed-size counterpart. The performance during an actual resize operation of course suffers somewhat due to the cache misses causes by the updates to each element's pointers, and this effect is most pronounced when the memory system becomes a bottleneck. This indicates that hash tables should be resized by substantial amounts, and that hysteresis should be applied to prevent

www.dbooks.org

performance degradation due to too-frequent resize operations. In memory-rich environments, hash-table sizes should furthermore be increased much more aggressively than they are decreased.

Another key point is that although the `hashtab` structure is non-partitionable, it is also read-mostly, which suggests the use of RCU. Given that the performance and scalability of this resizable hash table is very nearly that of RCU-protected fixed-sized hash tables, we must conclude that this approach was quite successful.

Finally, it is important to note that insertions, deletions, and lookups can proceed concurrently with a resize operation. This concurrency is critically important when resizing large hash tables, especially for applications that must meet severe response-time constraints.

Of course, the `ht_elem` structure's pair of pointer sets does impose some memory overhead, which is taken up in the next section.

10.4.4 Other Resizable Hash Tables

One shortcoming of the resizable hash table described earlier in this section is memory consumption. Each data element has two pairs of linked-list pointers rather than just one. Is it possible to create an RCU-protected resizable hash table that makes do with just one pair?

It turns out that the answer is "yes". Josh Triplett et al. [TMW11] produced a *relativistic hash table* that incrementally splits and combines corresponding hash chains so that readers always see valid hash chains at all points during the resizing operation. This incremental splitting and combining relies on the fact that it is harmless for a reader to see a data element that should be in some other hash chain: When this happens, the reader will simply ignore the extraneous data element due to key mismatches.

The process of shrinking a relativistic hash table by a factor of two is shown in Figure 10.20, in this case shrinking a two-bucket hash table into a one-bucket hash table, otherwise known as a linear list. This process works by coalescing pairs of buckets in the old larger hash table into single buckets in the new smaller hash table. For this process to work correctly, we clearly need to constrain the hash functions for the two tables. One such constraint is to use the same underlying hash function for both tables, but to throw out the low-order bit when shrinking from large to small. For example, the old two-bucket hash table would use the two top bits of the value, while the new one-bucket hash table could use the top bit of the value. In this way, a given pair of adjacent even and odd buckets

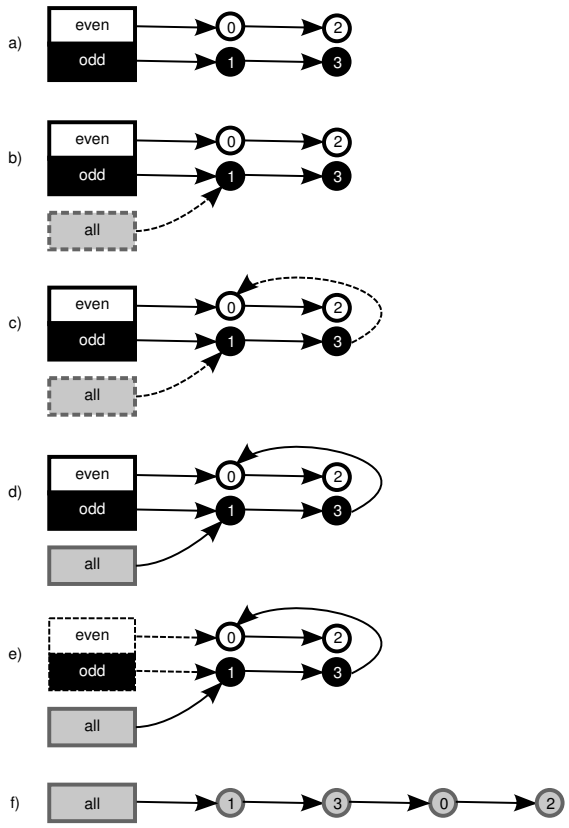


Figure 10.20: Shrinking a Relativistic Hash Table

in the old large hash table can be coalesced into a single bucket in the new small hash table, while still having a single hash value cover all of the elements in that single bucket.

The initial state is shown at the top of the figure, with time advancing from top to bottom, starting with initial state (a). The shrinking process begins by allocating the new smaller array of buckets, and having each bucket of this new smaller array reference the first element of one of the buckets of the corresponding pair in the old large hash table, resulting in state (b).

Then the two hash chains are linked together, resulting in state (c). In this state, readers looking up an even-numbered element see no change, and readers looking up elements 1 and 3 likewise see no change. However, readers looking up some other odd number will also traverse elements 0 and 2. This is harmless because any odd number will compare not-equal to these two elements. There is some performance loss, but on the other hand, this is exactly the same performance loss that will be

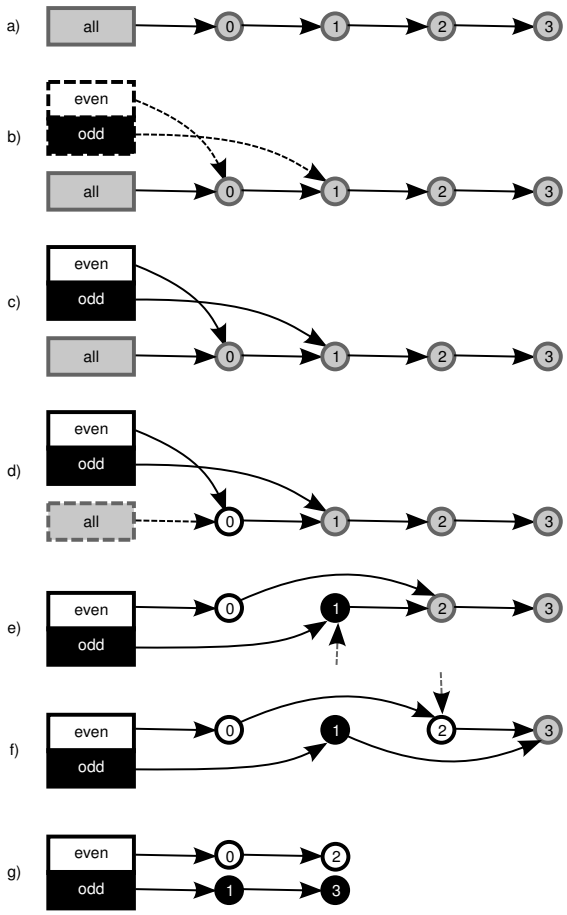


Figure 10.21: Growing a Relativistic Hash Table

experienced once the new small hash table is fully in place.

Next, the new small hash table is made accessible to readers, resulting in state (d). Note that older readers might still be traversing the old large hash table, so in this state both hash tables are in use.

The next step is to wait for all pre-existing readers to complete, resulting in state (e). In this state, all readers are using the new small hash table, so that the old large hash table's buckets may be freed, resulting in the final state (f).

Growing a relativistic hash table reverses the shrinking process, but requires more grace-period steps, as shown in Figure 10.21. The initial state (a) is at the top of this figure, with time advancing from top to bottom.

We start by allocating the new large two-bucket hash table, resulting in state (b). Note that each of these new buckets references the first element destined for that bucket. These new buckets are published to readers, resulting in state (c). After a grace-period operation, all readers are using the new large hash table, resulting in state (d). In this state, only those readers traversing the even-values hash bucket traverse element 0, which is therefore now colored white.

At this point, the old small hash buckets may be freed, although many implementations use these old buckets to track progress "unzipping" the list of items into their respective new buckets. The last even-numbered element in the first consecutive run of such elements now has its pointer-to-next updated to reference the following even-numbered element. After a subsequent grace-period operation, the result is state (e). The vertical arrow indicates the next element to be unzipped, and element 1 is now colored black to indicate that only those readers traversing the odd-values hash bucket may reach it.

Next, the last odd-numbered element in the first consecutive run of such elements now has its pointer-to-next updated to reference the following odd-numbered element. After a subsequent grace-period operation, the result is state (f). A final unzipping operation (including a grace-period operation) results in the final state (g).

In short, the relativistic hash table reduces the number of per-element list pointers at the expense of additional grace periods incurred during resizing. These additional grace periods are usually not a problem because insertions, deletions, and lookups may proceed concurrently with a resize operation.

It turns out that it is possible to reduce the per-element memory overhead from a pair of pointers to a single pointer, while still retaining $O(1)$ deletions. This is accomplished by augmenting split-order list [SS06] with RCU protection [Des09b, MDJ13a]. The data elements in the hash table are arranged into a single sorted linked list, with each hash bucket referencing the first element in that bucket. Elements are deleted by setting low-order bits in their pointer-to-next fields, and these elements are removed from the list by later traversals that encounter them.

This RCU-protected split-order list is complex, but offers lock-free progress guarantees for all insertion, deletion, and lookup operations. Such guarantees can be important in real-time applications. An implementation is available from recent versions of the userspace RCU library [Des09b].

www.dbooks.org

10.5 Other Data Structures

> All life is an experiment. The more experiments you make the better.
>
> *Ralph Waldo Emerson*

The preceding sections have focused on data structures that enhance concurrency due to partitionability (Section 10.2), efficient handling of read-mostly access patterns (Section 10.3), or application of read-mostly techniques to avoid non-partitionability (Section 10.4). This section gives a brief review of other data structures.

One of the hash table's greatest advantages for parallel use is that it is fully partitionable, at least while not being resized. One way of preserving the partitionability and the size independence is to use a radix tree, which is also called a trie. Tries partition the search key, using each successive key partition to traverse the next level of the trie. As such, a trie can be thought of as a set of nested hash tables, thus providing the required partitionability. One disadvantage of tries is that a sparse key space can result in inefficient use of memory. There are a number of compression techniques that may be used to work around this disadvantage, including hashing the key value to a smaller keyspace before the traversal [ON07]. Radix trees are heavily used in practice, including in the Linux kernel [Pig06].

One important special case of both a hash table and a trie is what is perhaps the oldest of data structures, the array and its multi-dimensional counterpart, the matrix. The fully partitionable nature of matrices is exploited heavily in concurrent numerical algorithms.

Self-balancing trees are heavily used in sequential code, with AVL trees and red-black trees being perhaps the most well-known examples [CLRS01]. Early attempts to parallelize AVL trees were complex and not necessarily all that efficient [Ell80], however, more recent work on red-black trees provides better performance and scalability by using RCU for readers and hashed arrays of locks[1] to protect reads and updates, respectively [HW11, HW13]. It turns out that red-black trees rebalance aggressively, which works well for sequential programs, but not necessarily so well for parallel use. Recent work has therefore made use of RCU-protected "bonsai trees" that rebalance less aggressively [CKZ12], trading off optimal tree depth to gain more efficient concurrent updates.

[1] In the guise of swissTM [DFGG11], which is a variant of software transactional memory in which the developer flags non-shared accesses.

Concurrent skip lists lend themselves well to RCU readers, and in fact represents an early academic use of a technique resembling RCU [Pug90].

Concurrent double-ended queues were discussed in Section 6.1.2, and concurrent stacks and queues have a long history [Tre86], though not normally the most impressive performance or scalability. They are nevertheless a common feature of concurrent libraries [MDJ13b]. Researchers have recently proposed relaxing the ordering constraints of stacks and queues [Sha11], with some work indicating that relaxed-ordered queues actually have better ordering properties than do strict FIFO queues [HKLP12, KLP12, HHK+13].

It seems likely that continued work with concurrent data structures will produce novel algorithms with surprising properties.

10.6 Micro-Optimization

> The devil is in the details.
>
> *Unknown*

The data structures shown in this section were coded straightforwardly, with no adaptation to the underlying system's cache hierarchy. In addition, many of the implementations used pointers to functions for key-to-hash conversions and other frequent operations. Although this approach provides simplicity and portability, in many cases it does give up some performance.

The following sections touch on specialization, memory conservation, and hardware considerations. Please do not mistake these short sections for a definitive treatise on this subject. Whole books have been written on optimizing to a specific CPU, let alone to the set of CPU families in common use today.

10.6.1 Specialization

The resizable hash table presented in Section 10.4 used an opaque type for the key. This allows great flexibility, permitting any sort of key to be used, but it also incurs significant overhead due to the calls via of pointers to functions. Now, modern hardware uses sophisticated branch-prediction techniques to minimize this overhead, but on the other hand, real-world software is often larger than can be accommodated even by today's large hardware branch-prediction tables. This is especially the case for calls via pointers, in which case the branch prediction

hardware must record a pointer in addition to branch-taken/branch-not-taken information.

This overhead can be eliminated by specializing a hash-table implementation to a given key type and hash function, for example, by using C++ templates. Doing so eliminates the `->ht_cmp()`, `->ht_gethash()`, and `->ht_getkey()` function pointers in the `ht` structure shown in Listing 10.9 on page 185. It also eliminates the corresponding calls through these pointers, which could allow the compiler to inline the resulting fixed functions, eliminating not only the overhead of the call instruction, but the argument marshalling as well.

In addition, the resizable hash table is designed to fit an API that segregates bucket selection from concurrency control. Although this allows a single torture test to exercise all the hash-table implementations in this chapter, it also means that many operations must compute the hash and interact with possible resize operations twice rather than just once. In a performance-conscious environment, the `hashtab_lock_mod()` function would also return a reference to the bucket selected, eliminating the subsequent call to `ht_get_bucket()`.

Quick Quiz 10.16: Couldn't the `hashtorture.h` code be modified to accommodate a version of `hashtab_lock_mod()` that subsumes the `ht_get_bucket()` functionality? ■

Quick Quiz 10.17: How much do these specializations really save? Are they really worth it? ■

All that aside, one of the great benefits of modern hardware compared to that available when I first started learning to program back in the early 1970s is that much less specialization is required. This allows much greater productivity than was possible back in the days of four-kilobyte address spaces.

10.6.2 Bits and Bytes

The hash tables discussed in this chapter made almost no attempt to conserve memory. For example, the `->ht_idx` field in the `ht` structure in Listing 10.9 on page 185 always has a value of either zero or one, yet takes up a full 32 bits of memory. It could be eliminated, for example, by stealing a bit from the `->ht_resize_key` field. This works because the `->ht_resize_key` field is large enough to address every byte of memory and the `ht_bucket` structure is more than one byte long, so that the `->ht_resize_key` field must have several bits to spare.

This sort of bit-packing trick is frequently used in data structures that are highly replicated, as is the `page` structure in the Linux kernel. However, the resizable hash table's `ht` structure is not all that highly replicated. It is instead the `ht_bucket` structures we should focus on. There are two major opportunities for shrinking the `ht_bucket` structure: (1) Placing the `->htb_lock` field in a low-order bit of one of the `->htb_head` pointers and (2) Reducing the number of pointers required.

The first opportunity might make use of bit-spinlocks in the Linux kernel, which are provided by the `include/linux/bit_spinlock.h` header file. These are used in space-critical data structures in the Linux kernel, but are not without their disadvantages:

1. They are significantly slower than the traditional spinlock primitives.
2. They cannot participate in the lockdep deadlock detection tooling in the Linux kernel [Cor06a].
3. They do not record lock ownership, further complicating debugging.
4. They do not participate in priority boosting in -rt kernels, which means that preemption must be disabled when holding bit spinlocks, which can degrade real-time latency.

Despite these disadvantages, bit-spinlocks are extremely useful when memory is at a premium.

One aspect of the second opportunity was covered in Section 10.4.4, which presented resizable hash tables that require only one set of bucket-list pointers in place of the pair of sets required by the resizable hash table presented in Section 10.4. Another approach would be to use singly linked bucket lists in place of the doubly linked lists used in this chapter. One downside of this approach is that deletion would then require additional overhead, either by marking the outgoing pointer for later removal or by searching the bucket list for the element being deleted.

In short, there is a tradeoff between minimal memory overhead on the one hand, and performance and simplicity on the other. Fortunately, the relatively large memories available on modern systems have allowed us to prioritize performance and simplicity over memory overhead. However, even with today's large-memory systems[2] it is sometimes necessary to take extreme measures to reduce memory overhead.

[2] Smartphones with gigabytes of memory, anyone?

www.dbooks.org

Listing 10.14: Alignment for 64-Byte Cache Lines

```
1 struct hash_elem {
2   struct ht_elem e;
3   long __attribute__ ((aligned(64))) counter;
4 };
```

10.6.3 Hardware Considerations

Modern computers typically move data between CPUs and main memory in fixed-sized blocks that range in size from 32 bytes to 256 bytes. These blocks are called *cache lines*, and are extremely important to high performance and scalability, as was discussed in Section 3.2. One timeworn way to kill both performance and scalability is to place incompatible variables into the same cacheline. For example, suppose that a resizable hash table data element had the `ht_elem` structure in the same cacheline as a frequently incremented counter. The frequent incrementing would cause the cacheline to be present at the CPU doing the incrementing, but nowhere else. If other CPUs attempted to traverse the hash bucket list containing that element, they would incur expensive cache misses, degrading both performance and scalability.

One way to solve this problem on systems with 64-byte cache line is shown in Listing 10.14. Here GCC's `aligned` attribute is used to force the `->counter` and the `ht_elem` structure into separate cache lines. This would allow CPUs to traverse the hash bucket list at full speed despite the frequent incrementing.

Of course, this raises the question "How did we know that cache lines are 64 bytes in size?" On a Linux system, this information may be obtained from the `/sys/devices/system/cpu/cpu*/cache/` directories, and it is even possible to make the installation process rebuild the application to accommodate the system's hardware structure. However, this would be more difficult if you wanted your application to also run on non-Linux systems. Furthermore, even if you were content to run only on Linux, such a self-modifying installation poses validation challenges. For example, systems with 32-byte cachelines might work well, but performance might suffer on systems with 64-byte cachelines due to false sharing.

Fortunately, there are some rules of thumb that work reasonably well in practice, which were gathered into a 1995 paper [GKPS95].[3] The first group of rules involve rearranging structures to accommodate cache geometry:

1. Place read-mostly data far from frequently updated data. For example, place read-mostly data at the beginning of the structure and frequently updated data at the end. Place data that is rarely accessed in between.

2. If the structure has groups of fields such that each group is updated by an independent code path, separate these groups from each other. Again, it can be helpful to place rarely accessed data between the groups. In some cases, it might also make sense to place each such group into a separate structure referenced by the original structure.

3. Where possible, associate update-mostly data with a CPU, thread, or task. We saw several very effective examples of this rule of thumb in the counter implementations in Chapter 5.

4. Going one step further, partition your data on a per-CPU, per-thread, or per-task basis, as was discussed in Chapter 8.

There has been some work towards automated trace-based rearrangement of structure fields [GDZE10]. This work might well ease one of the more painstaking tasks required to get excellent performance and scalability from multithreaded software.

An additional set of rules of thumb deal with locks:

1. Given a heavily contended lock protecting data that is frequently modified, take one of the following approaches:
   (a) Place the lock in a different cacheline than the data that it protects.
   (b) Use a lock that is adapted for high contention, such as a queued lock.
   (c) Redesign to reduce lock contention. (This approach is best, but is not always trivial.)

2. Place uncontended locks into the same cache line as the data that they protect. This approach means that the cache miss that brings the lock to the current CPU also brings its data.

3. Protect read-mostly data with hazard pointers, RCU, or, for long-duration critical sections, reader-writer locks.

Of course, these are rules of thumb rather than absolute rules. Some experimentation is required to work out which are most applicable to a given situation.

[3] A number of these rules are paraphrased and expanded on here with permission from Orran Krieger.

10.7 Summary

> There's only one thing more painful than learning from experience, and that is not learning from experience.
>
> *Archibald MacLeish*

This chapter has focused primarily on hash tables, including resizable hash tables, which are not fully partitionable. Section 10.5 gave a quick overview of a few non-hash-table data structures. Nevertheless, this exposition of hash tables is an excellent introduction to the many issues surrounding high-performance scalable data access, including:

1. Fully partitioned data structures work well on small systems, for example, single-socket systems.
2. Larger systems require locality of reference as well as full partitioning.
3. Read-mostly techniques, such as hazard pointers and RCU, provide good locality of reference for read-mostly workloads, and thus provide excellent performance and scalability even on larger systems.
4. Read-mostly techniques also work well on some types of non-partitionable data structures, such as resizable hash tables.
5. Large data structures can overflow CPU caches, reducing performance and scalability.
6. Additional performance and scalability can be obtained by specializing the data structure to a specific workload, for example, by replacing a general key with a 32-bit integer.
7. Although requirements for portability and for extreme performance often conflict, there are some data-structure-layout techniques that can strike a good balance between these two sets of requirements.

That said, performance and scalability are of little use without reliability, so the next chapter covers validation.

www.dbooks.org

If it is not tested, it doesn't work.

Unknown

Chapter 11

Validation

I have had a few parallel programs work the first time, but that is only because I have written a large number parallel programs over the past three decades. And I have had far more parallel programs that fooled me into thinking that they were working correctly the first time than actually were working the first time.

I thus need to validate my parallel programs. The basic trick behind validation, is to realize that the computer knows what is wrong. It is therefore your job to force it to tell you. This chapter can therefore be thought of as a short course in machine interrogation. But you can leave the good-cop/bad-cop routine at home. This chapter covers much more sophisticated and effective methods, especially given that most computers couldn't tell a good cop from a bad cop, at least as far as we know.

A longer course may be found in many recent books on validation, as well as at least one older but valuable one [Mye79]. Validation is an extremely important topic that cuts across all forms of software, and is worth intensive study in its own right. However, this book is primarily about concurrency, so this chapter will do little more than scratch the surface of this critically important topic.

Section 11.1 introduces the philosophy of debugging. Section 11.2 discusses tracing, Section 11.3 discusses assertions, and Section 11.4 discusses static analysis. Section 11.5 describes some unconventional approaches to code review that can be helpful when the fabled 10,000 eyes happen not to be looking at your code. Section 11.6 overviews the use of probability for validating parallel software. Because performance and scalability are first-class requirements for parallel programming, Section 11.7 covers these topics. Finally, Section 11.8 gives a fanciful summary and a short list of statistical traps to avoid.

But never forget that the two best debugging tools are a thorough understanding of the requirements, a solid design and a good night's sleep!

11.1 Introduction

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

Edsger W. Dijkstra

Section 11.1.1 discusses the sources of bugs, and Section 11.1.2 overviews the mindset required when validating software. Section 11.1.3 discusses when you should start validation, and Section 11.1.4 describes the surprisingly effective open-source regimen of code review and community testing.

11.1.1 Where Do Bugs Come From?

Bugs come from developers. The basic problem is that the human brain did not evolve with computer software in mind. Instead, the human brain evolved in concert with other human brains and with animal brains. Because of this history, the following three characteristics of computers often come as a shock to human intuition:

1. Computers lack common sense, despite huge sacrifices at the altar of artificial intelligence.
2. Computers fail to understand user intent, or more formally, computers generally lack a theory of mind.
3. Computers cannot do anything useful with a fragmentary plan, instead requiring that every detail of all possible scenarios be spelled out in full.

The first two points should be uncontroversial, as they are illustrated by any number of failed products, perhaps most famously Clippy and Microsoft Bob. By attempting to relate to users as people, these two products raised

Edition.2-rc11

www.dbooks.org

common-sense and theory-of-mind expectations that they proved incapable of meeting. Perhaps the set of software assistants are now available on smartphones will fare better, but as of 2020 reviews are mixed. That said, the developers working on them by all accounts still develop the old way: The assistants might well benefit end users, but not so much their own developers.

This human love of fragmentary plans deserves more explanation, especially given that it is a classic two-edged sword. This love of fragmentary plans is apparently due to the assumption that the person carrying out the plan will have (1) common sense and (2) a good understanding of the intent and requirements driving the plan. This latter assumption is especially likely to hold in the common case where the person doing the planning and the person carrying out the plan are one and the same: In this case, the plan will be revised almost subconsciously as obstacles arise, especially when that person has the a good understanding of the problem at hand. In fact, the love of fragmentary plans has served human beings well, in part because it is better to take random actions that have a some chance of locating food than to starve to death while attempting to plan the unplannable. However, the usefulness of fragmentary plans in everyday life is the everyday life of which we are all experts is no guarantee of their future usefulness in stored-program computers.

Furthermore, the need to follow fragmentary plans has had important effects on the human psyche, due to the fact that throughout much of human history, life was often difficult and dangerous. It should come as no surprise that executing a fragmentary plan that has a high probability of a violent encounter with sharp teeth and claws requires almost insane levels of optimism—a level of optimism that actually is present in most human beings. These insane levels of optimism extend to self-assessments of programming ability, as evidenced by the effectiveness of (and the controversy over) code-interviewing techniques [Bra07]. In fact, the clinical term for a human being with less-than-insane levels of optimism is "clinically depressed." Such people usually have extreme difficulty functioning in their daily lives, underscoring the perhaps counter-intuitive importance of insane levels of optimism to a normal, healthy life. Furtheremore, if you are not insanely optimistic, you are less likely to start a difficult but worthwhile project.[1]

[1] There are some famous exceptions to this rule of thumb. Some people take on difficult or risky projects in order to at least a temporarily escape from their depression. Others have nothing to lose: the project is literally a matter of life or death.

Quick Quiz 11.1: When in computing is it necessary to follow a fragmentary plan? ■

An important special case is the project that, while valuable, is not valuable enough to justify the time required to implement it. This special case is quite common, and one early symptom is the unwillingness of the decision-makers to invest enough to actually implement the project. A natural reaction is for the developers to produce an unrealistically optimistic estimate in order to be permitted to start the project. If the organization is strong enough and its decision-makers ineffective enough, the project might succeed despite the resulting schedule slips and budget overruns. However, if the organization is not strong enough and if the decision-makers fail to cancel the project as soon as it becomes clear that the estimates are garbage, then the project might well kill the organization. This might result in another organization picking up the project and either completing it, canceling it, or being killed by it. A given project might well succeed only after killing several organizations. One can only hope that the organization that eventually makes a success of a serial-organization-killer project maintains a suitable level of humility, lest it be killed by its next such project.

Quick Quiz 11.2: Who cares about the organization? After all, it is the project that is important! ■

Important though insane levels of optimism might be, they are a key source of bugs (and perhaps failure of organizations). The question is therefore "How to maintain the optimism required to start a large project while at the same time injecting enough reality to keep the bugs down to a dull roar?" The next section examines this conundrum.

11.1.2 Required Mindset

When carrying out any validation effort, keep the following definitions firmly in mind:

1. The only bug-free programs are trivial programs.
2. A reliable program has no known bugs.

From these definitions, it logically follows that any reliable non-trivial program contains at least one bug that you do not know about. Therefore, any validation effort undertaken on a non-trivial program that fails to find any bugs is itself a failure. A good validation is therefore an exercise in destruction. This means that if you are the

type of person who enjoys breaking things, validation is just job for you.

Quick Quiz 11.3: Suppose that you are writing a script that processes the output of the `time` command, which looks as follows:

```
real    0m0.132s
user    0m0.040s
sys     0m0.008s
```

The script is required to check its input for errors, and to give appropriate diagnostics if fed erroneous `time` output. What test inputs should you provide to this program to test it for use with `time` output generated by single-threaded programs? ■

But perhaps you are a super-programmer whose code is always perfect the first time every time. If so, congratulations! Feel free to skip this chapter, but I do hope that you will forgive my skepticism. You see, I have too many people who claimed to be able to write perfect code the first time, which is not too surprising given the previous discussion of optimism and over-confidence. And even if you really are a super-programmer, you just might find yourself debugging lesser mortals' work.

One approach for the rest of us is to alternate between our normal state of insane optimism (Sure, I can program that!) and severe pessimism (It seems to work, but I just know that there have to be more bugs hiding in there somewhere!). It helps if you enjoy breaking things. If you don't, or if your joy in breaking things is limited to breaking *other* people's things, find someone who does love breaking your code and have them help you break it.

Another helpful frame of mind is to hate it when other people find bugs in your code. This hatred can help motivate you to torture your code beyond all reason in order to increase the probability that you will be the one to find the bugs. Just make sure to suspend this hatred long enough to sincerely thank anyone who does find a bug in your code! After all, by so doing, they saved you the trouble of tracking it down, and possibly at great personal expense dredging through your code.

Yet another helpful frame of mind is studied skepticism. You see, believing that you understand the code means you can learn absolutely nothing about it. Ah, but you know that you completely understand the code because you wrote or reviewed it? Sorry, but the presence of bugs suggests that your understanding is at least partially fallacious. One cure is to write down what you know to be true and double-check this knowledge, as discussed in Sections 11.2–11.5. Objective reality *always* overrides whatever you might think you know.

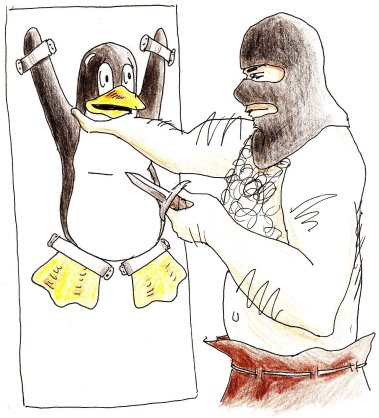


Figure 11.1: Validation and the Geneva Convention

One final frame of mind is to consider the possibility that someone's life depends on your code being correct. One way of looking at this is that consistently making good things happen requires a lot of focus on a lot of bad things that might happen, with an eye towards preventing or otherwise handling those bad things.[2] The prospect of these bad things might also motivate you to torture your code into revealing the whereabouts of its bugs.

This wide variety of frames of mind opens the door to the possibility of multiple people with different frames of mind contributing to the project, with varying levels of optimism. This can work well, if properly organized.

Some people might see vigorous validation as a form of torture, as depicted in Figure 11.1.[3] Such people might do well to remind themselves that, Tux cartoons aside, they are really torturing an inanimate object, as shown in Figure 11.2. Rest assured that those who fail to torture their code are doomed to be tortured by it!

However, this leaves open the question of exactly when during the project lifetime validation should start, a topic taken up by the next section.

11.1.3 When Should Validation Start?

Validation should start exactly when the project starts.

To see this, consider that tracking down a bug is much harder in a large program than in a small one. Therefore, to minimize the time and effort required to track down

[2] For more on this philosophy, see the chapter entitled "The Power of Negative Thinking" from Chris Hadfield's excellent book entitled "An Astronaut's Guide to Life on Earth."

[3] The cynics among us might question whether these people are afraid that validation will find bugs that they will then be required to fix.

www.dbooks.org

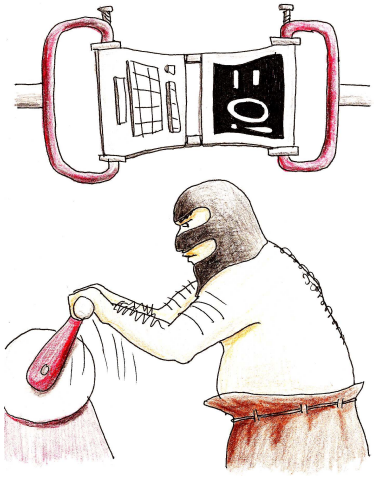


Figure 11.2: Rationalizing Validation

bugs, you should test small units of code. Although you won't find all the bugs this way, you will find a substantial fraction, and it will be much easier to find and fix the ones you do find. Testing at this level can also alert you to larger flaws in your overall design, minimizing the time you waste writing code that is broken by design.

But why wait until you have code before validating your design?[4] Hopefully reading Chapters 3 and 4 provided you with the information required to avoid some regrettably common design flaws, but discussing your design with a colleague or even simply writing it down can help flush out additional flaws.

However, it is all too often the case that waiting to start validation until you have a design is waiting too long. Mightn't your natural level of optimism caused you to start the design before you fully understood the requirements? The answer to this question will almost always be "yes". One good way to avoid flawed requirements is to get to know your users. To really serve them well, you will have to live among them.

Quick Quiz 11.4: You are asking me to do all this validation BS before I even start coding??? That sounds like a great way to never get started!!! ■

First-of-a-kind projects often use different methodologies such as rapid prototyping or agile. Here, the main goal of early prototypes are not to create correct implementations, but rather to learn the project's requirements. But this does not mean that you omit validation; it instead means that you approach it differently.

One such approach takes a Darwinian view, with the validation suite eliminating code that is not fit to solve the problem at hand. From this viewpoint, a vigorous validation suite is essential to the fitness of your software. However, taking this approach to its logical conclusion is quite humbling, as it requires us developers to admit that our carefully crafted changes to the codebase are, from a Darwinian standpoint, random mutations. On the other hand, this conclusion is supported by long experience indicating that seven percent of fixes introduce at least one bug [BJ12].

How vigorous should your validation suite be? If the bugs it finds aren't threatening the very foundations of your software design, then it is not yet vigorous enough. After all, your design is just as prone to bugs as is your code, and the earlier you find and fix the bugs in your design, the less time you will waste coding those design bugs.

Quick Quiz 11.5: Are you actually suggesting that it is possible to test correctness into software??? Everyone knows that is impossible!!! ■

It is worth reiterating that this advice applies to first-of-a-kind projects. If you are instead doing a project in a well-explored area, you would be quite foolish to refuse to learn from previous experience. But you should still start validating right at the beginning of the project, but hopefully guided by others' hard-won knowledge of both requirements and pitfalls.

An equally important question is "When should validation stop?" The best answer is "Some time after the last change." Every change has the potential to create a bug, and thus every change must be validated. Furthermore, validation development should continue through the full lifetime of the project. After all, the Darwinian perspective above implies that bugs are adapting to your validation suite. Therefore, unless you continually improve your validation suite, your project will naturally accumulate hordes of validation-suite-immune bugs.

But life is a tradeoff, and every bit of time invested in validation suites as a bit of time that cannot be invested in directly improving the project itself. These sorts of choices are never easy, and it can be just as damaging to overinvest in validation as it can be to underinvest. But this is just one more indication that life is not easy.

Now that we have established that you should start validation when you start the project (if not earlier!), and that both validation and validation development should

[4] The old saying "First we must code, then we have incentive to think" notwithstanding.

continue throughout the lifetime of that project, the following sections cover a number of validation techniques and methods that have proven their worth.

11.1.4 The Open Source Way

The open-source programming methodology has proven quite effective, and includes a regimen of intense code review and testing.

I can personally attest to the effectiveness of the open-source community's intense code review. One of my first patches to the Linux kernel involved a distributed filesystem where one node might write to a given file that another node has mapped into memory. In this case, it is necessary to invalidate the affected pages from the mapping in order to allow the filesystem to maintain coherence during the write operation. I coded up a first attempt at a patch, and, in keeping with the open-source maxim "post early, post often", I posted the patch. I then considered how I was going to test it.

But before I could even decide on an overall test strategy, I got a reply to my posting pointing out a few bugs. I fixed the bugs and reposted the patch, and returned to thinking out my test strategy. However, before I had a chance to write any test code, I received a reply to my reposted patch, pointing out more bugs. This process repeated itself many times, and I am not sure that I ever got a chance to actually test the patch.

This experience brought home the truth of the open-source saying: Given enough eyeballs, all bugs are shallow [Ray99].

However, when you post some code or a given patch, it is worth asking a few questions:

1. How many of those eyeballs are actually going to look at your code?
2. How many will be experienced and clever enough to actually find your bugs?
3. Exactly when are they going to look?

I was lucky: There was someone out there who wanted the functionality provided by my patch, who had long experience with distributed filesystems, and who looked at my patch almost immediately. If no one had looked at my patch, there would have been no review, and therefore none of those bugs would have been located. If the people looking at my patch had lacked experience with distributed filesystems, it is unlikely that they would have found all the bugs. Had they waited months or even years to look, I likely would have forgotten how the patch was supposed to work, making it much more difficult to fix them.

However, we must not forget the second tenet of the open-source development, namely intensive testing. For example, a great many people test the Linux kernel. Some test patches as they are submitted, perhaps even yours. Others test the -next tree, which is helpful, but there is likely to be several weeks or even months delay between the time that you write the patch and the time that it appears in the -next tree, by which time the patch will not be quite as fresh in your mind. Still others test maintainer trees, which often have a similar time delay.

Quite a few people don't test code until it is committed to mainline, or the master source tree (Linus's tree in the case of the Linux kernel). If your maintainer won't accept your patch until it has been tested, this presents you with a deadlock situation: your patch won't be accepted until it is tested, but it won't be tested until it is accepted. Nevertheless, people who test mainline code are still relatively aggressive, given that many people and organizations do not test code until it has been pulled into a Linux distro.

And even if someone does test your patch, there is no guarantee that they will be running the hardware and software configuration and workload required to locate your bugs.

Therefore, even when writing code for an open-source project, you need to be prepared to develop and run your own test suite. Test development is an underappreciated and very valuable skill, so be sure to take full advantage of any existing test suites available to you. Important as test development is, we must leave further discussion of it to books dedicated to that topic. The following sections therefore discuss locating bugs in your code given that you already have a good test suite.

11.2 Tracing

> The machine knows what is wrong. Make it tell you.
>
> *Unknown*

When all else fails, add a `printk()`! Or a `printf()`, if you are working with user-mode C-language applications.

The rationale is simple: If you cannot figure out how execution reached a given point in the code, sprinkle print statements earlier in the code to work out what happened. You can get a similar effect, and with more convenience and flexibility, by using a debugger such as gdb (for user applications) or kgdb (for debugging Linux kernels).

www.dbooks.org

Much more sophisticated tools exist, with some of the more recent offering the ability to rewind backwards in time from the point of failure.

These brute-force testing tools are all valuable, especially now that typical systems have more than 64K of memory and CPUs running faster than 4 MHz. Much has been written about these tools, so this chapter will add only a little more.

However, these tools all have a serious shortcoming when you need a fastpath to tell you what is going wrong, namely, these tools often have excessive overheads. There are special tracing technologies for this purpose, which typically leverage data ownership techniques (see Chapter 8) to minimize the overhead of runtime data collection. One example within the Linux kernel is "trace events" [Ros10b, Ros10c, Ros10d, Ros10a], which uses per-CPU buffers to allow data to be collected with extremely low overhead. Even so, enabling tracing can sometimes change timing enough to hide bugs, resulting in *heisenbugs*, which are discussed in Section 11.6 and especially Section 11.6.4. In the kernel, BPF can do data reduction in the kernel, reducing the overhead of transmitting the needed information from the kernel to userspace [Gre19]. In userspace code, there is a huge number of tools that can help you. One good starting point is Brendan Gregg's blog.[5]

Even if you avoid heisenbugs, other pitfalls await you. For example, although the machine really does know all, what it knows is almost always way more than your head can hold. For this reason, high-quality test suites normally come with sophisticated scripts to analyze the voluminous output. But beware—scripts will only notice what you tell them to. My rcutorture scripts are a case in point: Early versions of those scripts were quite satisfied with a test run in which RCU grace periods stalled indefinitely. This of course resulted in the scripts being modified to detect RCU grace-period stalls, but this does not change the fact that the scripts will only detect problems that I make them detect. But note well that unless you have a solid design, you won't know what your script should check for!

Another problem with tracing and especially with `printk()` calls is that their overhead rules out production use. In some such cases, assertions can be helpful.

[5] `http://www.brendangregg.com/blog/`

11.3 Assertions

> No man really becomes a fool until he stops asking questions.
>
> *Charles P. Steinmetz*

Assertions are usually implemented in the following manner:

```
1 if (something_bad_is_happening())
2   complain();
```

This pattern is often encapsulated into C-preprocessor macros or language intrinsics, for example, in the Linux kernel, this might be represented as `WARN_ON(something_bad_is_happening())`. Of course, if `something_bad_is_happening()` quite frequently, the resulting output might obscure reports of other problems, in which case `WARN_ON_ONCE(something_bad_is_happening())` might be more appropriate.

Quick Quiz 11.6: How can you implement `WARN_ON_ONCE()`? ■

In parallel code, one bad something that might happen is that a function expecting to be called under a particular lock might be called without that lock being held. Such functions sometimes have header comments stating something like "The caller must hold `foo_lock` when calling this function", but such a comment does no good unless someone actually reads it. An executable statement carries far more weight. The Linux kernel's lockdep facility [Cor06a, Ros11] therefore provides a `lockdep_assert_held()` function that checks whether the proper locks are held. Of course, lockdep incurs significant overhead, and thus not necessarily helpful in production.

An especially bad parallel-code something is unexpected concurrent access to data. The Kernel Concurrency Sanitizer (KCSAN) [Cor16a] uses existing markings such as `READ_ONCE()` and `WRITE_ONCE()` to determine which concurrent accesses deserve warning messages. KCSAN has a significant false-positive rate, especially in from the viewpoint of developers thinking in terms of C as assembly language with additional syntax. KCSAN therefore provides a `data_race()` construct to forgive known-benign data races, and also the `ASSERT_EXCLUSIVE_ACCESS()` and `ASSERT_EXCLUSIVE_WRITER()` assertions to explicitly check for data races[EMV+20a, EMV+20b].

So what can be done in cases where checking is necessary, but where the overhead of runtime checking cannot

be tolerated? One approach is static analysis, which is discussed in the next section.

11.4 Static Analysis

> A lot of automation isn't a replacement of humans but of mind-numbing behavior.
>
> *Summarized from Stewart Butterfield*

Static analysis is a validation technique where one program takes a second program as input, reporting errors and vulnerabilities located in this second program. Interestingly enough, almost all programs are statically analyzed by their compilers or interpreters. These tools are far from perfect, but their ability to locate errors has improved immensely over the past few decades, in part because they now have much more than 64K bytes of memory in which to carry out their analyses.

The original UNIX `lint` tool [Joh77] was quite useful, though much of its functionality has since been incorporated into C compilers. There are nevertheless lint-like tools in use to this day. The sparse static analyzer [Cor04b] finds higher-level issues in the Linux kernel, including:

1. Misuse of pointers to user-space structures.
2. Assignments from too-long constants.
3. Empty `switch` statements.
4. Mismatched lock acquisition and release primitives.
5. Misuse of per-CPU primitives.
6. Use of RCU primitives on non-RCU pointers and vice versa.

Although it is likely that compilers will continue to increase their static-analysis capabilities, the sparse static analyzer demonstrates the benefits of static analysis outside of the compiler, particularly for finding application-specific bugs. Sections 12.4–12.5 describe more sophisticated forms of static analysis.

11.5 Code Review

> If a man speaks of my virtues, he steals from me; if he speaks of my vices, then he is my teacher.
>
> *Chinese proverb*

Code review isa special case of static analysis with human beings doing the analysis. This section covers inspection, walkthroughs, and self-inspection.

11.5.1 Inspection

Traditionally, formal code inspections take place in face-to-face meetings with formally defined roles: moderator, developer, and one or two other participants. The developer reads through the code, explaining what it is doing and why it works. The one or two other participants ask questions and raise issues, hopefully exposing the author's invalid assumptions, while the moderator's job is to resolve any resulting conflicts and take notes. This process can be extremely effective at locating bugs, particularly if all of the participants are familiar with the code at hand.

However, this face-to-face formal procedure does not necessarily work well in the global Linux kernel community. Instead, individuals review code separately and provide comments via email or IRC. The note-taking is provided by email archives or IRC logs, and moderators volunteer their services as required by the occasional flamewar. This process also works reasonably well, particularly if all of the participants are familiar with the code at hand. In fact, one advantage of the Linux kernel community approach over traditional formal inspections is the greater probability of contributions from people *not* familiar with the code, who might not be blinded by the author's invalid assumptions, and who might also test the code.

Quick Quiz 11.7: Just what invalid assumptions are you accusing Linux kernel hackers of harboring??? ■

It is quite likely that the Linux kernel community's review process is ripe for improvement:

1. There is sometimes a shortage of people with the time and expertise required to carry out an effective review.
2. Even though all review discussions are archived, they are often "lost" in the sense that insights are forgotten and people fail to look up the discussions. This can result in re-insertion of the same old bugs.

www.dbooks.org

3. It is sometimes difficult to resolve flamewars when they do break out, especially when the combatants have disjoint goals, experience, and vocabulary.

Perhaps some of the needed improvements will be provided by continuous-integration-style testing, but there are many bugs more easily found by review than by testing. When reviewing, therefore, it is worthwhile to look at relevant documentation in commit logs, bug reports, and LWN articles. This documentation can help you quickly build up the required expertise.

11.5.2 Walkthroughs

A traditional code walkthrough is similar to a formal inspection, except that the group "plays computer" with the code, driven by specific test cases. A typical walkthrough team has a moderator, a secretary (who records bugs found), a testing expert (who generates the test cases) and perhaps one to two others. These can be extremely effective, albeit also extremely time-consuming.

It has been some decades since I have participated in a formal walkthrough, and I suspect that a present-day walkthrough would use single-stepping debuggers. One could imagine a particularly sadistic procedure as follows:

1. The tester presents the test case.
2. The moderator starts the code under a debugger, using the specified test case as input.
3. Before each statement is executed, the developer is required to predict the outcome of the statement and explain why this outcome is correct.
4. If the outcome differs from that predicted by the developer, this is taken as a potential bug.
5. In parallel code, a "concurrency shark" asks what code might execute concurrently with this code, and why such concurrency is harmless.

Sadistic, certainly. Effective? Maybe. If the participants have a good understanding of the requirements, software tools, data structures, and algorithms, then walkthroughs can be extremely effective. If not, walkthroughs are often a waste of time.

11.5.3 Self-Inspection

Although developers are usually not all that effective at inspecting their own code, there are a number of situations where there is no reasonable alternative. For example, the developer might be the only person authorized to look at the code, other qualified developers might all be too busy, or the code in question might be sufficiently bizarre that the developer is unable to convince anyone else to take it seriously until after demonstrating a prototype. In these cases, the following procedure can be quite helpful, especially for complex parallel code:

1. Write design document with requirements, diagrams for data structures, and rationale for design choices.
2. Consult with experts, updating the design document as needed.
3. Write the code in pen on paper, correcting errors as you go. Resist the temptation to refer to pre-existing nearly identical code sequences, instead, copy them.
4. At each step, articulate and question your assumptions, inserting assertions or constructing tests to check them.
5. If there were errors, copy the code in pen on fresh paper, correcting errors as you go. Repeat until the last two copies are identical.
6. Produce proofs of correctness for any non-obvious code.
7. Use a source-code control system. Commit early; commit often.
8. Test the code fragments from the bottom up.
9. When all the code is integrated (but preferably before), do full-up functional and stress testing.
10. Once the code passes all tests, write code-level documentation, perhaps as an extension to the design document discussed above. Fix both the code and the test code as needed.

When I follow this procedure for new RCU code, there are normally only a few bugs left at the end. With a few prominent (and embarrassing) exceptions [McK11a], I usually manage to locate these bugs before others do. That said, this is getting more difficult over time as the number and variety of Linux-kernel users increases.

Quick Quiz 11.8: Why would anyone bother copying existing code in pen on paper??? Doesn't that just increase the probability of transcription errors? ■

Quick Quiz 11.9: This procedure is ridiculously over-engineered! How can you expect to get a reasonable amount of software written doing it this way??? ■

Quick Quiz 11.10: What do you do if, after all the pen-on-paper copying, you find a bug while typing in the resulting code? ■

The above procedure works well for new code, but what if you need to inspect code that you have already written? You can of course apply the above procedure for old code in the special case where you wrote one to throw away [FPB79], but the following approach can also be helpful in less desperate circumstances:

1. Using your favorite documentation tool (LaTeX, HTML, OpenOffice, or straight ASCII), describe the high-level design of the code in question. Use lots of diagrams to illustrate the data structures and how these structures are updated.
2. Make a copy of the code, stripping away all comments.
3. Document what the code does statement by statement.
4. Fix bugs as you find them.

This works because describing the code in detail is an excellent way to spot bugs [Mye79]. This second procedure is also a good way to get your head around someone else's code, in many cases, the first step suffices.

Although review and inspection by others is probably more efficient and effective, the above procedures can be quite helpful in cases where for whatever reason it is not feasible to involve others.

At this point, you might be wondering how to write parallel code without having to do all this boring paperwork. Here are some time-tested ways of accomplishing this:

1. Write a sequential program that scales through use of available parallel library functions.
2. Write sequential plug-ins for a parallel framework, such as map-reduce, BOINC, or a web-application server.
3. Fully partition your problems, then implement sequential program(s) that run in parallel without communication.
4. Stick to one of the application areas (such as linear algebra) where tools can automatically decompose and parallelize the problem.

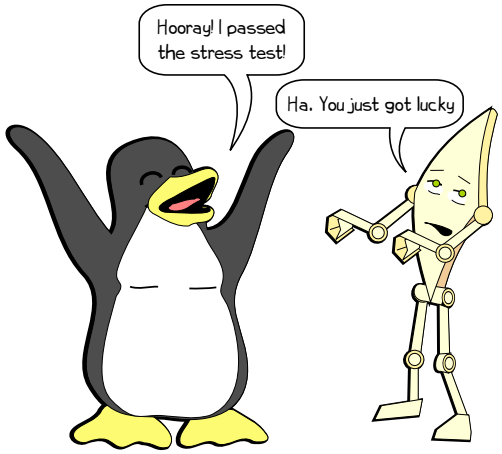


Figure 11.3: Passed on Merits? Or Dumb Luck?

5. Make extremely disciplined use of parallel-programming primitives, so that the resulting code is easily seen to be correct. But beware: It is always tempting to break the rules "just a little bit" to gain better performance or scalability. Breaking the rules often results in general breakage. That is, unless you carefully do the paperwork described in this section.

But the sad fact is that even if you do the paperwork or use one of the above ways to more-or-less safely avoid paperwork, there will be bugs. If nothing else, more users and a greater variety of users will expose more bugs more quickly, especially if those users are doing things that the original developers did not consider. The next section describes how to handle the probabilistic bugs that occur all too commonly when validating parallel software.

Quick Quiz 11.11: Wait! Why on earth would an abstract piece of software fail only sometimes??? ■

11.6 Probability and Heisenbugs

> With both heisenbugs and impressionist art, the closer you get, the less you see.
>
> *Unknown*

So your parallel program fails sometimes. But you used techniques from the earlier sections to locate the problem and now have a fix in place! Congratulations!!!

www.dbooks.org

Now the question is just how much testing is required in order to be certain that you actually fixed the bug, as opposed to just reducing the probability of it occurring on the one hand, having fixed only one of several related bugs on the other hand, or made some ineffectual unrelated change on yet a third hand. In short, what is the answer to the eternal question posed by Figure 11.3?

Unfortunately, the honest answer is that an infinite amount of testing is required to attain absolute certainty.

Quick Quiz 11.12: Suppose that you had a very large number of systems at your disposal. For example, at current cloud prices, you can purchase a huge amount of CPU time at low cost. Why not use this approach to get close enough to certainty for all practical purposes? ■

But suppose that we are willing to give up absolute certainty in favor of high probability. Then we can bring powerful statistical tools to bear on this problem. However, this section will focus on simple statistical tools. These tools are extremely helpful, but please note that reading this section is not a substitute for statistics classes.[6]

For our start with simple statistical tools, we need to decide whether we are doing discrete or continuous testing. Discrete testing features well-defined individual test runs. For example, a boot-up test of a Linux kernel patch is an example of a discrete test: The kernel either comes up or it does not. Although you might spend an hour boot-testing your kernel, the number of times you attempted to boot the kernel and the number of times the boot-up succeeded would often be of more interest than the length of time you spent testing. Functional tests tend to be discrete.

On the other hand, if my patch involved RCU, I would probably run rcutorture, which is a kernel module that, strangely enough, tests RCU. Unlike booting the kernel, where the appearance of a login prompt signals the successful end of a discrete test, rcutorture will happily continue torturing RCU until either the kernel crashes or until you tell it to stop. The duration of the rcutorture test is usually of more interest than the number of times you started and stopped it. Therefore, rcutorture is an example of a continuous test, a category that includes many stress tests.

Statistics for discrete tests are simpler and more familiar than those for continuous tests, and furthermore the statistics for discrete tests can often be pressed into service for continuous tests, though with some loss of accuracy. We therefore start with discrete tests.

[6] Which I most highly recommend. The few statistics courses I have taken have provided value far beyond that of the time I spent on them.

11.6.1 Statistics for Discrete Testing

Suppose a bug has a 10 % chance of occurring in a given run and that we do five runs. How do we compute the probability of at least one run failing? Here is one way:

1. Compute the probability of a given run succeeding, which is 90 %.
2. Compute the probability of all five runs succeeding, which is 0.9 raised to the fifth power, or about 59 %.
3. Because either all five runs succeed, or at least one fails, subtract the 59 % expected success rate from 100 %, yielding a 41 % expected failure rate.

For those preferring formulas, call the probability of a single failure f. The probability of a single success is then $1 - f$ and the probability that all of n tests will succeed is S_n:

$$S_n = (1 - f)^n \tag{11.1}$$

The probability of failure is $1 - S_n$, or:

$$F_n = 1 - (1 - f)^n \tag{11.2}$$

Quick Quiz 11.13: Say what??? When I plug the earlier five-test 10 %-failure-rate example into the formula, I get 59,050 % and that just doesn't make sense!!! ■

So suppose that a given test has been failing 10 % of the time. How many times do you have to run the test to be 99 % sure that your supposed fix actually helped?

Another way to ask this question is "How many times would we need to run the test to cause the probability of failure to rise above 99 %?" After all, if we were to run the test enough times that the probability of seeing at least one failure becomes 99 %, if there are no failures, there is only 1 % probability of this "success" being due to dumb luck. And if we plug $f = 0.1$ into Equation 11.2 and vary n, we find that 43 runs gives us a 98.92 % chance of at least one test failing given the original 10 % per-test failure rate, while 44 runs gives us a 99.03 % chance of at least one test failing. So if we run the test on our fix 44 times and see no failures, there is a 99 % probability that our fix really did help.

But repeatedly plugging numbers into Equation 11.2 can get tedious, so let's solve for n:

$$F_n = 1 - (1 - f)^n \tag{11.3}$$

$$1 - F_n = (1 - f)^n \tag{11.4}$$

$$\log (1 - F_n) = n \log (1 - f) \tag{11.5}$$

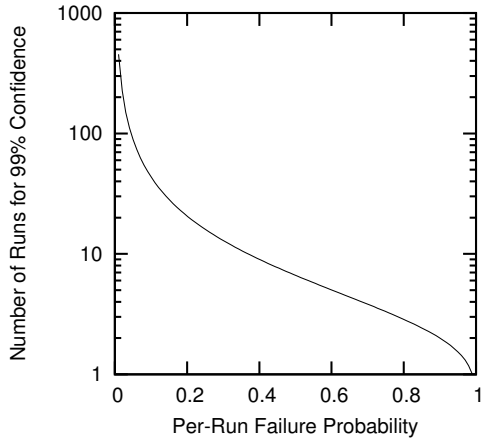


Figure 11.4: Number of Tests Required for 99 Percent Confidence Given Failure Rate

Finally the number of tests required is given by:

$$n = \frac{\log(1 - F_n)}{\log(1 - f)} \tag{11.6}$$

Plugging $f = 0.1$ and $F_n = 0.99$ into Equation 11.6 gives 43.7, meaning that we need 44 consecutive successful test runs to be 99 % certain that our fix was a real improvement. This matches the number obtained by the previous method, which is reassuring.

Quick Quiz 11.14: In Equation 11.6, are the logarithms base-10, base-2, or base-e? ■

Figure 11.4 shows a plot of this function. Not surprisingly, the less frequently each test run fails, the more test runs are required to be 99 % confident that the bug has been fixed. If the bug caused the test to fail only 1 % of the time, then a mind-boggling 458 test runs are required. As the failure probability decreases, the number of test runs required increases, going to infinity as the failure probability goes to zero.

The moral of this story is that when you have found a rarely occurring bug, your testing job will be much easier if you can come up with a carefully targeted test with a much higher failure rate. For example, if your targeted test raised the failure rate from 1 % to 30 %, then the number of runs required for 99 % confidence would drop from 458 to a more tractable 13.

But these thirteen test runs would only give you 99 % confidence that your fix had produced "some improvement". Suppose you instead want to have 99 % confidence that your fix reduced the failure rate by an order of magnitude. How many failure-free test runs are required?

An order of magnitude improvement from a 30 % failure rate would be a 3 % failure rate. Plugging these numbers into Equation 11.6 yields:

$$n = \frac{\log(1 - 0.99)}{\log(1 - 0.03)} = 151.2 \tag{11.7}$$

So our order of magnitude improvement requires roughly an order of magnitude more testing. Certainty is impossible, and high probabilities are quite expensive. This is why making tests run more quickly and making failures more probable are essential skills in the development of highly reliable software. These skills will be covered in Section 11.6.4.

11.6.2 Statistics Abuse for Discrete Testing

But suppose that you have a continuous test that fails about three times every ten hours, and that you fix the bug that you believe was causing the failure. How long do you have to run this test without failure to be 99 % certain that you reduced the probability of failure?

Without doing excessive violence to statistics, we could simply redefine a one-hour run to be a discrete test that has a 30 % probability of failure. Then the results of in the previous section tell us that if the test runs for 13 hours without failure, there is a 99 % probability that our fix actually improved the program's reliability.

A dogmatic statistician might not approve of this approach, but the sad fact is that the errors introduced by this sort of statistical abuse are usually quite small compared to the errors in your failure-rate estimates. Nevertheless, the next section takes a more rigorous approach.

11.6.3 Statistics for Continuous Testing

The fundamental formula for failure probabilities is the Poisson distribution:

$$F_m = \frac{\lambda^m}{m!} e^{-\lambda} \tag{11.8}$$

Here F_m is the probability of m failures in the test and λ is the expected failure rate per unit time. A rigorous derivation may be found in any advanced probability textbook, for example, Feller's classic "An Introduction to Probability Theory and Its Applications" [Fel50], while a more intuitive derivation may be found in the first edition of this book [McK14a].

Let's try reworking the example from Section 11.6.2 using the Poisson distribution. Recall that this example involved a test with a 30 % failure rate per hour, and that

www.dbooks.org

the question was how long the test would need to run error-free on a alleged fix to be 99 % certain that the fix actually reduced the failure rate. In this case, m is zero, so that Equation 11.8 reduces to:

$$F_0 = e^{-\lambda} \tag{11.9}$$

Solving this requires setting F_0 to 0.01 and solving for λ, resulting in:

$$\lambda = -\ln 0.01 = 4.6 \tag{11.10}$$

Because we get 0.3 failures per hour, the number of hours required is $4.6/0.3 = 14.3$, which is within 10 % of the 13 hours calculated using the method in Section 11.6.2. Given that you normally won't know your failure rate to anywhere near 10 %, the simpler method described in Section 11.6.2 is almost always good and sufficient.

More generally, if we have n failures per unit time, and we want to be P % certain that a fix reduced the failure rate, we can use the following formula:

$$T = -\frac{1}{n} \ln \frac{100 - P}{100} \tag{11.11}$$

Quick Quiz 11.15: Suppose that a bug causes a test failure three times per hour on average. How long must the test run error-free to provide 99.9 % confidence that the fix significantly reduced the probability of failure? ■

As before, the less frequently the bug occurs and the greater the required level of confidence, the longer the required error-free test run.

Suppose that a given test fails about once every hour, but after a bug fix, a 24-hour test run fails only twice. Assuming that the failure leading to the bug is a random occurrence, what is the probability that the small number of failures in the second run was due to random chance? In other words, how confident should we be that the fix actually had some effect on the bug? This probability may be calculated by summing Equation 11.8 as follows:

$$F_0 + F_1 + \ldots + F_{m-1} + F_m = \sum_{i=0}^{m} \frac{\lambda^i}{i!} e^{-\lambda} \tag{11.12}$$

This is the Poisson cumulative distribution function, which can be written more compactly as:

$$F_{i \le m} = \sum_{i=0}^{m} \frac{\lambda^i}{i!} e^{-\lambda} \tag{11.13}$$

Here m is the actual number of errors in the long test run (in this case, two) and λ is expected number of errors in the long test run (in this case, 24). Plugging $m = 2$ and $\lambda = 24$ into this expression gives the probability of two or fewer failures as about 1.2×10^{-8}, in other words, we have a high level of confidence that the fix actually had some relationship to the bug.[7]

Quick Quiz 11.16: Doing the summation of all the factorials and exponentials is a real pain. Isn't there an easier way? ■

Quick Quiz 11.17: But wait!!! Given that there has to be *some* number of failures (including the possibility of zero failures), shouldn't Equation 11.13 approach the value 1 as m goes to infinity? ■

The Poisson distribution is a powerful tool for analyzing test results, but the fact is that in this last example there were still two remaining test failures in a 24-hour test run. Such a low failure rate results in very long test runs. The next section discusses counter-intuitive ways of improving this situation.

11.6.4 Hunting Heisenbugs

This line of thought also helps explain heisenbugs: adding tracing and assertions can easily reduce the probability of a bug appearing, which is why extremely lightweight tracing and assertion mechanism are so critically important.

The term "heisenbug" was inspired by the Heisenberg Uncertainty Principle from quantum physics, which states that it is impossible to exactly quantify a given particle's position and velocity at any given point in time [Hei27]. Any attempt to more accurately measure that particle's position will result in increased uncertainty of its velocity and vice versa. Similarly, attempts to track down the heisenbug causes its symptoms to radically change or even disappear completely.[8]

If the field of physics inspired the name of this problem, it is only fair that the field of physics should inspire the solution. Fortunately, particle physics is up to the task: Why not create an anti-heisenbug to annihilate the heisenbug? Or, perhaps more accurately, to annihilate the heisen-ness of the heisenbug? Although producing an anti-heisenbug for a given heisenbug is more an art than a

[7] Of course, this result in no way excuses you from finding and fixing the bug(s) resulting in the remaining two failures!

[8] The term "heisenbug" is a misnomer, as most heisenbugs are fully explained by the *observer effect* from classical physics. Nevertheless, the name has stuck.

science, the following sections describe a number of ways to do just that:

1. Add delay to race-prone regions (Section 11.6.4.1).
2. Increase workload intensity (Section 11.6.4.2).
3. Isolate suspicious subsystems (Section 11.6.4.3).
4. Simulate unusual events (Section 11.6.4.4).
5. Count near misses (Section 11.6.4.5).

These are followed by discussion in Section 11.6.4.6.

11.6.4.1 Add Delay

Consider the count-lossy code in Section 5.1. Adding `printf()` statements will likely greatly reduce or even eliminate the lost counts. However, converting the load-add-store sequence to a load-add-delay-store sequence will greatly increase the incidence of lost counts (try it!). Once you spot a bug involving a race condition, it is frequently possible to create an anti-heisenbug by adding delay in this manner.

Of course, this begs the question of how to find the race condition in the first place. This is a bit of a dark art, but there are a number of things you can do to find them.

One approach is to recognize that race conditions often end up corrupting some of the data involved in the race. It is therefore good practice to double-check the synchronization of any corrupted data. Even if you cannot immediately recognize the race condition, adding delay before and after accesses to the corrupted data might change the failure rate. By adding and removing the delays in an organized fashion (e.g., binary search), you might learn more about the workings of the race condition.

Quick Quiz 11.18: How is this approach supposed to help if the corruption affected some unrelated pointer, which then caused the corruption??? ■

Another important approach is to vary the software and hardware configuration and look for statistically significant differences in failure rate. You can then look more intensively at the code implicated by the software or hardware configuration changes that make the greatest difference in failure rate. It might be helpful to test that code in isolation, for example.

One important aspect of software configuration is the history of changes, which is why `git bisect` is so useful. Bisection of the change history can provide very valuable clues as to the nature of the heisenbug.

Quick Quiz 11.19: But I did the bisection, and ended up with a huge commit. What do I do now? ■

However you locate the suspicious section of code, you can then introduce delays to attempt to increase the probability of failure. As we have seen, increasing the probability of failure makes it much easier to gain high confidence in the corresponding fix.

However, it is sometimes quite difficult to track down the problem using normal debugging techniques. The following sections present some other alternatives.

11.6.4.2 Increase Workload Intensity

It is often the case that a given test suite places relatively low stress on a given subsystem, so that a small change in timing can cause a heisenbug to disappear. One way to create an anti-heisenbug for this case is to increase the workload intensity, which has a good chance of increasing the bug's probability. If the probability is increased sufficiently, it may be possible to add lightweight diagnostics such as tracing without causing the bug to vanish.

How can you increase the workload intensity? This depends on the program, but here are some things to try:

1. Add more CPUs.
2. If the program uses networking, add more network adapters and more or faster remote systems.
3. If the program is doing heavy I/O when the problem occurs, either (1) add more storage devices, (2) use faster storage devices, for example, substitute SSDs for disks, or (3) use a RAM-based filesystem to substitute main memory for mass storage.
4. Change the size of the problem, for example, if doing a parallel matrix multiply, change the size of the matrix. Larger problems may introduce more complexity, but smaller problems often increase the level of contention. If you aren't sure whether you should go large or go small, just try both.

However, it is often the case that the bug is in a specific subsystem, and the structure of the program limits the amount of stress that can be applied to that subsystem. The next section addresses this situation.

11.6.4.3 Isolate Suspicious Subsystems

If the program is structured such that it is difficult or impossible to apply much stress to a subsystem that is

www.dbooks.org

under suspicion, a useful anti-heisenbug is a stress test that tests that subsystem in isolation. The Linux kernel's rcutorture module takes exactly this approach with RCU: Applying more stress to RCU than is feasible in a production environment increases the probability that RCU bugs will be found during testing rather than in production.[9]

In fact, when creating a parallel program, it is wise to stress-test the components separately. Creating such component-level stress tests can seem like a waste of time, but a little bit of component-level testing can save a huge amount of system-level debugging.

11.6.4.4 Simulate Unusual Events

Heisenbugs are sometimes due to unusual events, such as memory-allocation failure, conditional-lock-acquisition failure, CPU-hotplug operations, timeouts, packet losses, and so on. One way to construct an anti-heisenbug for this class of heisenbug is to introduce spurious failures.

For example, instead of invoking `malloc()` directly, invoke a wrapper function that uses a random number to decide whether to return `NULL` unconditionally on the one hand, or to actually invoke `malloc()` and return the resulting pointer on the other. Inducing spurious failures is an excellent way to bake robustness into sequential programs as well as parallel programs.

Quick Quiz 11.20: Why don't conditional-locking primitives provide this spurious-failure functionality? ■

11.6.4.5 Count Near Misses

Bugs are often all-or-nothing things, so that a bug either happens or not, with nothing in between. However, it is sometimes possible to define a *near miss* where the bug does not result in a failure, but has likely manifested. For example, suppose your code is making a robot walk. The robot's falling down constitutes a bug in your program, but stumbling and recovering might constitute a near miss. If the robot falls over only once per hour, but stumbles every few minutes, you might be able to speed up your debugging progress by counting the number of stumbles in addition to the number of falls.

In concurrent programs, timestamping can sometimes be used to detect near misses. For example, locking primitives incur significant delays, so if there is a too-short delay between a pair of operations that are supposed to be protected by different acquisitions of the same lock, this too-short delay might be counted as a near miss.[10]

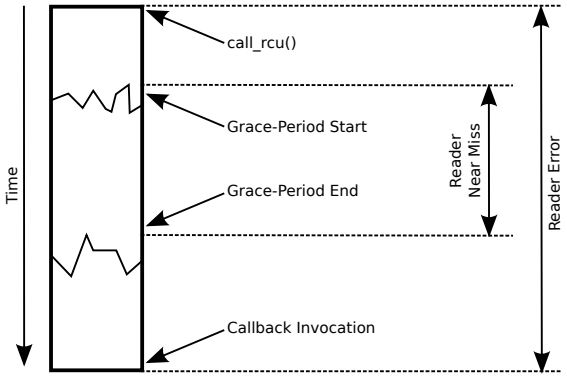


Figure 11.5: RCU Errors and Near Misses

For example, a low-probability bug in RCU priority boosting occurred roughly once every hundred hours of focused rcutorture testing. Because it would take almost 500 hours of failure-free testing to be 99 % certain that the bug's probability had been significantly reduced, the `git bisect` process to find the failure would be painfully slow—or would require an extremely large test farm. Fortunately, the RCU operation being tested included not only a wait for an RCU grace period, but also a previous wait for the grace period to start and a subsequent wait for an RCU callback to be invoked after completion of the RCU grace period. This distinction between an `rcutorture` error and near miss is shown in Figure 11.5. To qualify as a full-fledged error, an RCU read-side critical section must extend from the `call_rcu()` that initiated a grace period, through the remainder of the previous grace period, through the entirety of the grace period initiated by the `call_rcu()` (denoted by the region between the jagged lines), and through the delay from the end of that grace period to the callback invocation, as indicated by the "Error" arrow. However, the formal definition of RCU prohibits RCU read-side critical sections from extending across a single grace period, as indicated by the "Near Miss" arrow. This suggests using near misses as the error condition, however, this can be problematic because different CPUs can have different opinions as to exactly where a given grace period starts and ends, as indicated

[9] Though sadly not increased to probability one.

[10] Of course, in this case, you might be better off using whatever `lock_held()` primitive is available in your environment. If there isn't a `lock_held()` primitive, create one!

by the jagged lines.[11] Using the near misses as the error condition could therefore result in false positives, which need to be avoided in the automated `rcutorture` testing.

By sheer dumb luck, `rcutorture` happens to include some statistics that are sensitive to the near-miss version of the grace period. As noted above, these statistics are subject to false positives due to their unsynchronized access to RCU's state variables, but these false positives turn out to be extremely rare on strongly ordered systems such as the IBM mainframe and x86, occurring less than once per thousand hours of testing.

These near misses occurred roughly once per hour, about two orders of magnitude more frequently than the actual errors. Use of these near misses allowed the bug's root cause to be identified in less than a week and a high degree of confidence in the fix to be built in less than a day. In contrast, excluding the near misses in favor of the real errors would have required months of debug and validation time.

To sum up near-miss counting, the general approach is to replace counting of infrequent failures with more-frequent near misses that are believed to be correlated with those failures. These near-misses can be considered an anti-heisenbug to the real failure's heisenbug because the near-misses, being more frequent, are likely to be more robust in the face of changes to your code, for example, the changes you make to add debugging code.

11.6.4.6 Heisenbug Discussion

The alert reader might have noticed that this section was fuzzy and qualitative, in stark contrast to the precise mathematics of Sections 11.6.1, 11.6.2, and 11.6.3. If you love precision and mathematics, you may be disappointed to learn that the situations to which this section applies are far more common than those to which the preceding sections apply.

In fact, the common case is that although you might have reason to believe that your code has bugs, you have no idea what those bugs are, what causes them, how likely they are to appear, or what conditions affect their probability of appearance. In this all-too-common case, statistics cannot help you.[12] That is to say, statistics cannot help you *directly*. But statistics can be of great indirect help—*if* you have the humility required to admit that you make mistakes, that you can reduce the probability of these mistakes (for example, by getting enough sleep), and that the number and type of mistakes you made in the past is indicative of the number and type of mistakes that you are likely to make in the future. For example, I have a deplorable tendency to forget to write a small but critical portion of the initialization code, and frequently get most or even all of a parallel program correct—except for a stupid omission in initialization. Once I was willing to admit to myself that I am prone to this type of mistake, it was easier (but not easy!) to force myself to double-check my initialization code. Doing this allowed me to find numerous bugs ahead of time.

Using Taleb's nomenclature [Tal07], a white swan is a bug that we can reproduce. We can run a large number of tests, use ordinary statistics to estimate the bug's probability, and use ordinary statistics again to estimate our confidence in a proposed fix. An unsuspected bug is a black swan. We know nothing about it, we have no tests that have yet caused it to happen, and statistics is of no help. Studying our own behavior, especially the number and types of mistakes we make, can turn black swans into grey swans. We might not know exactly what the bugs are, but we have some idea of their number and maybe also of their type. Ordinary statistics is still of no help (at least not until we are able to reproduce one of the bugs), but robust[13] testing methods can be of great help. The goal, therefore, is to use experience and good validation practices to turn the black swans grey, focused testing and analysis to turn the grey swans white, and ordinary methods to fix the white swans.

That said, thus far, we have focused solely on bugs in the parallel program's functionality. However, because performance is a first-class requirement for a parallel program (otherwise, why not write a sequential program?), the next section discusses performance bugs.

11.7 Performance Estimation

> There are lies, damn lies, statistics, and benchmarks.
>
> *Unknown*

Parallel programs usually have performance and scalability requirements, after all, if performance is not an issue, why not use a sequential program? Ultimate performance and linear scalability might not be necessary, but there is

[11] In real life, these lines can be much more jagged because idle CPUs can be completely unaware of a great many recent grace periods.

[12] Although if you know what your program is supposed to do and if your program is small enough (both less likely that you might think), then the formal-verification tools described in Chapter 12 can be helpful.

[13] That is to say brutal.

www.dbooks.org

little use for a parallel program that runs slower than its optimal sequential counterpart. And there really are cases where every microsecond matters and every nanosecond is needed. Therefore, for parallel programs, insufficient performance is just as much a bug as is incorrectness.

Quick Quiz 11.21: That is ridiculous!!! After all, isn't getting the correct answer later than one would like better than getting an incorrect answer??? ■

Quick Quiz 11.22: But if you are going to put in all the hard work of parallelizing an application, why not do it right? Why settle for anything less than optimal performance and linear scalability? ■

Validating a parallel program must therfore include validating its performance. But validating performance means having a workload to run and performance criteria with which to evaluate the program at hand. These needs are often met by *performance benchmarks*, which are discussed in the next section.

11.7.1 Benchmarking

Frequent abuse aside, benchmarks are both useful and heavily used, so it is not helpful to be too dismissive of them. Benchmarks span the range from ad hoc test jigs to international standards, but regardless of their level of formality, benchmarks serve four major purposes:

1. Providing a fair framework for comparing competing implementations.
2. Focusing competitive energy on improving implementations in ways that matter to users.
3. Serving as example uses of the implementations being benchmarked.
4. Serving as a marketing tool to highlight your software against your competitors' offerings.

Of course, the only completely fair framework is the intended application itself. So why would anyone who cared about fairness in benchmarking bother creating imperfect benchmarks rather than simply using the application itself as the benchmark?

Running the actual application is in fact the best approach where it is practical. Unfortunately, it is often impractical for the following reasons:

1. The application might be proprietary, and you might not have the right to run the intended application.
2. The application might require more hardware than you have access to.
3. The application might use data that you cannot access, for example, due to privacy regulations.
4. The application might take longer than is convenient to reproduce a performance or scalability problem.[14]

Creating a benchmark that approximates the application can help overcome these obstacles. A carefully constructed benchmark can help promote performance, scalability, energy efficiency, and much else besides. However, be careful to avoid investing too much into the benchmarking effort. It is after all important to invest at least a little into the application itself [Gra91].

11.7.2 Profiling

In many cases, a fairly small portion of your software is responsible for the majority of the performance and scalability shortfall. However, developers are notoriously unable to identify the actual bottlenecks by inspection. For example, in the case of a kernel buffer allocator, all attention focused on a search of a dense array which turned out to represent only a few percent of the allocator's execution time. An execution profile collected via a logic analyzer focused attention on the cache misses that were actually responsible for the majority of the problem [MS93].

An old-school but quite effective method of tracking down performance and scalability bugs is to run your program under a debugger, then periodically interrupt it, recording the stacks of all threads at each interruption. The theory here is that if something is slowing down your program, it has to be visible in your threads' executions.

That said, there are a number of tools that will usually do a much better job of helping you to focus your attention where it will do the most good. Two popular choices are `gprof` and `perf`. To use `perf` on a single-process program, prefix your command with `perf record`, then after the command completes, type `perf report`. There is a lot of work on tools for performance debugging of multi-threaded programs, which should make this important job easier. Again, one good starting point is Brendan Gregg's blog.[15]

[14] Microbenchmarks can help, but please see Section 11.7.4.
[15] `http://www.brendangregg.com/blog/`

11.7.3 Differential Profiling

Scalability problems will not necessarily be apparent unless you are running on very large systems. However, it is sometimes possible to detect impending scalability problems even when running on much smaller systems. One technique for doing this is called *differential profiling*.

The idea is to run your workload under two different sets of conditions. For example, you might run it on two CPUs, then run it again on four CPUs. You might instead vary the load placed on the system, the number of network adapters, the number of mass-storage devices, and so on. You then collect profiles of the two runs, and mathematically combine corresponding profile measurements. For example, if your main concern is scalability, you might take the ratio of corresponding measurements, and then sort the ratios into descending numerical order. The prime scalability suspects will then be sorted to the top of the list [McK95, McK99].

Some tools such as `perf` have built-in differential-profiling support.

11.7.4 Microbenchmarking

Microbenchmarking can be useful when deciding which algorithms or data structures are worth incorporating into a larger body of software for deeper evaluation.

One common approach to microbenchmarking is to measure the time, run some number of iterations of the code under test, then measure the time again. The difference between the two times divided by the number of iterations gives the measured time required to execute the code under test.

Unfortunately, this approach to measurement allows any number of errors to creep in, including:

1. The measurement will include some of the overhead of the time measurement. This source of error can be reduced to an arbitrarily small value by increasing the number of iterations.

2. The first few iterations of the test might incur cache misses or (worse yet) page faults that might inflate the measured value. This source of error can also be reduced by increasing the number of iterations, or it can often be eliminated entirely by running a few warm-up iterations before starting the measurement period. Most systems have ways of detecting whether a given process incurred a page fault, and you should make use of this to reject runs whose performance has been thus impeded.

3. Some types of interference, for example, random memory errors, are so rare that they can be dealt with by running a number of sets of iterations of the test. If the level of interference was statistically significant, any performance outliers could be rejected statistically.

4. Any iteration of the test might be interfered with by other activity on the system. Sources of interference include other applications, system utilities and daemons, device interrupts, firmware interrupts (including system management interrupts, or SMIs), virtualization, memory errors, and much else besides. Assuming that these sources of interference occur randomly, their effect can be minimized by reducing the number of iterations.

5. Thermal throttling can understate scalability because increasing CPU activity increases heat generation, and on systems without adequate cooling (most of them!), this can result in the CPU frequency decreasing as the number of CPUs increases.[16] Of course, if you are testing an application to evaluate its expected behavior when run in production, such thermal throttling is simply a fact of life. Otherwise, if you are interested in theoretical scalability, use a system with adequate cooling or reduce the CPU clock rate to a level that the cooling system can handle.

The first and fourth sources of interference provide conflicting advice, which is one sign that we are living in the real world. The remainder of this section looks at ways of resolving this conflict.

Quick Quiz 11.23: But what about other sources of error, for example, due to interactions between caches and memory layout? ■

The following sections discuss ways of dealing with these measurement errors, with Section 11.7.5 covering isolation techniques that may be used to prevent some forms of interference, and with Section 11.7.6 covering methods for detecting interference so as to reject measurement data that might have been corrupted by that interference.

11.7.5 Isolation

The Linux kernel provides a number of ways to isolate a group of CPUs from outside interference.

[16] Systems with adequate cooling tend to look like gaming systems.

www.dbooks.org

First, let's look at interference by other processes, threads, and tasks. The POSIX `sched_setaffinity()` system call may be used to move most tasks off of a given set of CPUs and to confine your tests to that same group. The Linux-specific user-level `taskset` command may be used for the same purpose, though both `sched_setaffinity()` and `taskset` require elevated permissions. Linux-specific control groups (cgroups) may be used for this same purpose. This approach can be quite effective at reducing interference, and is sufficient in many cases. However, it does have limitations, for example, it cannot do anything about the per-CPU kernel threads that are often used for housekeeping tasks.

One way to avoid interference from per-CPU kernel threads is to run your test at a high real-time priority, for example, by using the POSIX `sched_setscheduler()` system call. However, note that if you do this, you are implicitly taking on responsibility for avoiding infinite loops, because otherwise your test can prevent part of the kernel from functioning. This is an example of the Spiderman Principle: "With great power comes great responsibility." And although the default real-time throttling settings often address such problems, they might do so by causing your real-time threads to miss their deadlines.

These approaches can greatly reduce, and perhaps even eliminate, interference from processes, threads, and tasks. However, it does nothing to prevent interference from device interrupts, at least in the absence of threaded interrupts. Linux allows some control of threaded interrupts via the `/proc/irq` directory, which contains numerical directories, one per interrupt vector. Each numerical directory contains `smp_affinity` and `smp_affinity_list`. Given sufficient permissions, you can write a value to these files to restrict interrupts to the specified set of CPUs. For example, either "`echo 3 > /proc/irq/23/smp_affinity`" or "`echo 0-1 > /proc/irq/23/smp_affinity_list`" would confine interrupts on vector 23 to CPUs 0 and 1, at least given sufficient privileges. You can use "`cat /proc/interrupts`" to obtain a list of the interrupt vectors on your system, how many are handled by each CPU, and what devices use each interrupt vector.

Running a similar command for all interrupt vectors on your system would confine interrupts to CPUs 0 and 1, leaving the remaining CPUs free of interference. Or mostly free of interference, anyway. It turns out that the scheduling-clock interrupt fires on each CPU that is running in user mode.[17] In addition you must take care to ensure that the set of CPUs that you confine the interrupts to is capable of handling the load.

But this only handles processes and interrupts running in the same operating-system instance as the test. Suppose that you are running the test in a guest OS that is itself running on a hypervisor, for example, Linux running KVM? Although you can in theory apply the same techniques at the hypervisor level that you can at the guest-OS level, it is quite common for hypervisor-level operations to be restricted to authorized personnel. In addition, none of these techniques work against firmware-level interference.

Quick Quiz 11.24: Wouldn't the techniques suggested to isolate the code under test also affect that code's performance, particularly if it is running within a larger application? ■

Of course, if it is in fact the interference that is producing the behavior of interest, you will instead need to promote interference, in which case being unable to prevent it is not a problem. But if you really do need interference-free measurements, then instead of preventing the interference, you might need to detect the interference as described in the next section.

11.7.6 Detecting Interference

If you cannot prevent interference, perhaps you can detect it and reject results from any affected test runs. Section 11.7.6.1 describes methods of rejection involving additional measurements, while Section 11.7.6.2 describes statistics-based rejection.

11.7.6.1 Detecting Interference Via Measurement

Many systems, including Linux, provide means for determining after the fact whether some forms of interference have occurred. For example, process-based interference results in context switches, which, on Linux-based systems, are visible in `/proc/<PID>/sched` via the `nr_switches` field. Similarly, interrupt-based interference can be detected via the `/proc/interrupts` file.

Opening and reading files is not the way to low overhead, and it is possible to get the count of context switches for a given thread by using the `getrusage()` system call, as shown in Listing 11.1. This same system call can be used to detect minor page faults (`ru_minflt`) and major page faults (`ru_majflt`).

[17] Frederic Weisbecker leads up a `NO_HZ_FULL` adaptive-ticks project that allows scheduling-clock interrupts to be disabled on CPUs that have only one runnable task. As of 2021, this is largely complete.

Listing 11.1: Using `getrusage()` to Detect Context Switches

```
1  #include <sys/time.h>
2  #include <sys/resource.h>
3
4  /* Return 0 if test results should be rejected. */
5  int runtest(void)
6  {
7    struct rusage ru1;
8    struct rusage ru2;
9
10   if (getrusage(RUSAGE_SELF, &ru1) != 0) {
11     perror("getrusage");
12     abort();
13   }
14   /* run test here. */
15   if (getrusage(RUSAGE_SELF, &ru2 != 0) {
16     perror("getrusage");
17     abort();
18   }
19   return (ru1.ru_nvcsw == ru2.ru_nvcsw &&
20           ru1.runivcsw == ru2.runivcsw);
21 }
```

Unfortunately, detecting memory errors and firmware interference is quite system-specific, as is the detection of interference due to virtualization. Although avoidance is better than detection, and detection is better than statistics, there are times when one must avail oneself of statistics, a topic addressed in the next section.

11.7.6.2 Detecting Interference Via Statistics

Any statistical analysis will be based on assumptions about the data, and performance microbenchmarks often support the following assumptions:

1. Smaller measurements are more likely to be accurate than larger measurements.
2. The measurement uncertainty of good data is known.
3. A reasonable fraction of the test runs will result in good data.

The fact that smaller measurements are more likely to be accurate than larger measurements suggests that sorting the measurements in increasing order is likely to be productive.[18] The fact that the measurement uncertainty is known allows us to accept measurements within this uncertainty of each other: If the effects of interference are large compared to this uncertainty, this will ease rejection of bad data. Finally, the fact that some fraction (for example, one third) can be assumed to be good allows us to blindly accept the first portion of the sorted list, and this data can then be used to gain an estimate of the natural variation of the measured data, over and above the assumed measurement error.

The approach is to take the specified number of leading elements from the beginning of the sorted list, and use these to estimate a typical inter-element delta, which in turn may be multiplied by the number of elements in the list to obtain an upper bound on permissible values. The algorithm then repeatedly considers the next element of the list. If it falls below the upper bound, and if the distance between the next element and the previous element is not too much greater than the average inter-element distance for the portion of the list accepted thus far, then the next element is accepted and the process repeats. Otherwise, the remainder of the list is rejected.

Listing 11.2 shows a simple `sh/awk` script implementing this notion. Input consists of an x-value followed by an arbitrarily long list of y-values, and output consists of one line for each input line, with fields as follows:

1. The x-value.
2. The average of the selected data.
3. The minimum of the selected data.
4. The maximum of the selected data.
5. The number of selected data items.
6. The number of input data items.

This script takes three optional arguments as follows:

`--divisor`: Number of segments to divide the list into, for example, a divisor of four means that the first quarter of the data elements will be assumed to be good. This defaults to three.

`--relerr`: Relative measurement error. The script assumes that values that differ by less than this error are for all intents and purposes equal. This defaults to 0.01, which is equivalent to 1 %.

`--trendbreak`: Ratio of inter-element spacing constituting a break in the trend of the data. For example, if the average spacing in the data accepted so far is 1.5, then if the trend-break ratio is 2.0, then if the next data value differs from the last one by more than 3.0, this constitutes a break in the trend. (Unless of course, the relative error is greater than 3.0, in which case the "break" will be ignored.)

[18] To paraphrase the old saying, "Sort first and ask questions later."

www.dbooks.org

Listing 11.2: Statistical Elimination of Interference

```
1  div=3
2  rel=0.01
3  tre=10
4  while test $# -gt 0
5  do
6    case "$1" in
7    --divisor)
8      shift
9      div=$1
10     ;;
11   --relerr)
12     shift
13     rel=$1
14     ;;
15   --trendbreak)
16     shift
17     tre=$1
18     ;;
19   esac
20   shift
21 done
22
23 awk -v divisor=$div -v relerr=$rel -v trendbreak=$tre '{
24   for (i = 2; i <= NF; i++)
25     d[i - 1] = $i;
26   asort(d);
27   i = int((NF + divisor - 1) / divisor);
28   delta = d[i] - d[1];
29   maxdelta = delta * divisor;
30   maxdelta1 = delta + d[i] * relerr;
31   if (maxdelta1 > maxdelta)
32     maxdelta = maxdelta1;
33   for (j = i + 1; j < NF; j++) {
34     if (j <= 2)
35       maxdiff = d[NF - 1] - d[1];
36     else
37       maxdiff = trendbreak * (d[j - 1] - d[1]) / (j - 2);
38     if (d[j] - d[1] > maxdelta && d[j] - d[j - 1] > maxdiff)
39       break;
40   }
41   n = sum = 0;
42   for (k = 1; k < j; k++) {
43     sum += d[k];
44     n++;
45   }
46   min = d[1];
47   max = d[j - 1];
48   avg = sum / n;
49   print $1, avg, min, max, n, NF - 1;
50 }'
```

Lines 1–3 of Listing 11.2 set the default values for the parameters, and lines 4–21 parse any command-line overriding of these parameters. The `awk` invocation on line 23 sets the values of the `divisor`, `relerr`, and `trendbreak` variables to their `sh` counterparts. In the usual `awk` manner, lines 24–50 are executed on each input line. The loop spanning lines 24 and 25 copies the input y-values to the `d` array, which line 26 sorts into increasing order. Line 27 computes the number of trustworthy y-values by applying `divisor` and rounding up.

Lines 28–32 compute the `maxdelta` lower bound on the upper bound of y-values. To this end, line 29 multiplies the difference in values over the trusted region of data by the `divisor`, which projects the difference in values across the trusted region across the entire set of y-values. However, this value might well be much smaller than the relative error, so line 30 computes the absolute error (`d[i] * relerr`) and adds that to the difference `delta` across the trusted portion of the data. Lines 31 and 32 then compute the maximum of these two values.

Each pass through the loop spanning lines 33–40 attempts to add another data value to the set of good data. Lines 34–39 compute the trend-break delta, with line 34 disabling this limit if we don't yet have enough values to compute a trend, and with line 37 multiplying `trendbreak` by the average difference between pairs of data values in the good set. If line 38 determines that the candidate data value would exceed the lower bound on the upper bound (`maxdelta`) *and* that the difference between the candidate data value and its predecessor exceeds the trend-break difference (`maxdiff`), then line 39 exits the loop: We have the full good set of data.

Lines 41–49 then compute and print statistics.

Quick Quiz 11.25: This approach is just plain weird! Why not use means and standard deviations, like we were taught in our statistics classes? ■

Quick Quiz 11.26: But what if all the y-values in the trusted group of data are exactly zero? Won't that cause the script to reject any non-zero value? ■

Although statistical interference detection can be quite useful, it should be used only as a last resort. It is far better to avoid interference in the first place (Section 11.7.5), or, failing that, detecting interference via measurement (Section 11.7.6.1).

11.8 Summary

> To err is human! Stop being human!!!
>
> *Ed Nofziger*

Although validation never will be an exact science, much can be gained by taking an organized approach to it, as an organized approach will help you choose the right validation tools for your job, avoiding situations like the one fancifully depicted in Figure 11.6.

A key choice is that of statistics. Although the methods described in this chapter work very well most of the time, they do have their limitations, courtesy of the Halting Problem [Tur37, Pul00]. Fortunately for us, there is a

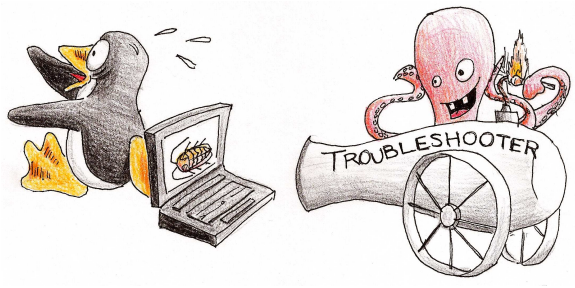


Figure 11.6: Choose Validation Methods Wisely

huge number of special cases in which we can not only work out whether a program will halt, but also estimate how long it will run before halting, as discussed in Section 11.7. Furthermore, in cases where a given program might or might not work correctly, we can often establish estimates for what fraction of the time it will work correctly, as discussed in Section 11.6.

Nevertheless, unthinking reliance on these estimates is brave to the point of foolhardiness. After all, we are summarizing a huge mass of complexity in code and data structures down to a single solitary number. Even though we can get away with such bravery a surprisingly large fraction of the time, abstracting all that code and data away will occasionally cause severe problems.

One possible problem is variability, where repeated runs give wildly different results. This problem is often addressed using standard deviation, however, using two numbers to summarize the behavior of a large and complex program is about as brave as using only one number. In computer programming, the surprising thing is that use of the mean or the mean and standard deviation are often sufficient. Nevertheless, there are no guarantees.

One cause of variation is confounding factors. For example, the CPU time consumed by a linked-list search will depend on the length of the list. Averaging together runs with wildly different list lengths will probably not be useful, and adding a standard deviation to the mean will not be much better. The right thing to do would be control for list length, either by holding the length constant or to measure CPU time as a function of list length.

Of course, this advice assumes that you are aware of the confounding factors, and Murphy says that you will not be. I have been involved in projects that had confounding factors as diverse as air conditioners (which drew considerable power at startup, thus causing the voltage supplied to the computer to momentarily drop too low, sometimes resulting in failure), cache state (resulting in odd variations in performance), I/O errors (including disk errors, packet loss, and duplicate Ethernet MAC addresses), and even porpoises (which could not resist playing with an array of transponders, which could be otherwise used for high-precision acoustic positioning and navigation). And this is but one reason why a good night's sleep is such an effective debugging tool.

In short, validation always will require some measure of the behavior of the system. To be at all useful, this measure must be a severe summarization of the system, which in turn means that it can be misleading. So as the saying goes, "Be careful. It is a real world out there."

But what if you are working on the Linux kernel, which as of 2017 was estimated to have more than 20 billion instances running throughout the world? In that case, a bug that occurs once every million years on a single system will be encountered more than 50 times per day across the installed base. A test with a 50 % chance of encountering this bug in a one-hour run would need to increase that bug's probability of occurrence by more than ten orders of magnitude, which poses a severe challenge to today's testing methodologies. One important tool that can sometimes be applied with good effect to such situations is formal verification, the subject of the next chapter, and, more speculatively, Section 17.4.

The topic of choosing a validation plan, be it testing, formal verification, or both, is taken up by Section 12.7.

www.dbooks.org

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth

Chapter 12

Formal Verification

Parallel algorithms can be hard to write, and even harder to debug. Testing, though essential, is insufficient, as fatal race conditions can have extremely low probabilities of occurrence. Proofs of correctness can be valuable, but in the end are just as prone to human error as is the original algorithm. In addition, a proof of correctness cannot be expected to find errors in your assumptions, shortcomings in the requirements, misunderstandings of the underlying software or hardware primitives, or errors that you did not think to construct a proof for. This means that formal methods can never replace testing. Nevertheless, formal methods can be a valuable addition to your validation toolbox.

It would be very helpful to have a tool that could somehow locate all race conditions. A number of such tools exist, for example, Section 12.1 provides an introduction to the general-purpose state-space search tools Promela and Spin, Section 12.2 similarly introduces the special-purpose ppcmem and cppmem tools, Section 12.3 looks at an example axiomatic approach, Section 12.4 briefly overviews SAT solvers, Section 12.5 briefly overviews stateless model checkers, Section 12.6 sums up use of formal-verification tools for verifying parallel algorithms, and finally Section 12.7 discusses how to decide how much and what type of validation to apply to a given software project.

12.1 State-Space Search

Follow every byway / Every path you know.

"Climb Every Mountain", Rodgers & Hammerstein

This section features the general-purpose Promela and Spin tools, which may be used to carry out a full state-space search of many types of multi-threaded code. They are used to verifying data communication protocols. Section 12.1.1 introduces Promela and Spin, including a couple of warm-up exercises verifying both non-atomic and atomic increment. Section 12.1.2 describes use of Promela, including example command lines and a comparison of Promela syntax to that of C. Section 12.1.3 shows how Promela may be used to verify locking, 12.1.4 uses Promela to verify an unusual implementation of RCU named "QRCU", and finally Section 12.1.5 applies Promela to early versions of RCU's dyntick-idle implementation.

12.1.1 Promela and Spin

Promela is a language designed to help verify protocols, but which can also be used to verify small parallel algorithms. You recode your algorithm and correctness constraints in the C-like language Promela, and then use Spin to translate it into a C program that you can compile and run. The resulting program carries out a full state-space search of your algorithm, either verifying or finding counter-examples for assertions that you can associate with in your Promela program.

This full-state search can be extremely powerful, but can also be a two-edged sword. If your algorithm is too complex or your Promela implementation is careless, there might be more states than fit in memory. Furthermore, even given sufficient memory, the state-space search might well run for longer than the expected lifetime of the universe. Therefore, use this tool for compact but complex parallel algorithms. Attempts to naively apply it to even moderate-scale algorithms (let alone the full Linux kernel) will end badly.

Promela and Spin may be downloaded from `https://spinroot.com/spin/whatispin.html`.

The above site also gives links to Gerard Holzmann's excellent book [Hol03] on Promela and Spin, as well as

Edition.2-rc11

www.dbooks.org

Listing 12.1: Promela Code for Non-Atomic Increment

```
1  #define NUMPROCS 2
2
3  byte counter = 0;
4  byte progress[NUMPROCS];
5
6  proctype incrementer(byte me)
7  {
8    int temp;
9
10   temp = counter;
11   counter = temp + 1;
12   progress[me] = 1;
13 }
14
15 init {
16   int i = 0;
17   int sum = 0;
18
19   atomic {
20     i = 0;
21     do
22     :: i < NUMPROCS ->
23       progress[i] = 0;
24       run incrementer(i);
25       i++;
26     :: i >= NUMPROCS -> break;
27     od;
28   }
29   atomic {
30     i = 0;
31     sum = 0;
32     do
33     :: i < NUMPROCS ->
34       sum = sum + progress[i];
35       i++
36     :: i >= NUMPROCS -> break;
37     od;
38     assert(sum < NUMPROCS || counter == NUMPROCS);
39   }
40 }
```

searchable online references starting at: `https://www.spinroot.com/spin/Man/index.html`.

The remainder of this section describes how to use Promela to debug parallel algorithms, starting with simple examples and progressing to more complex uses.

12.1.1.1 Warm-Up: Non-Atomic Increment

Listing 12.1 demonstrates the textbook race condition resulting from non-atomic increment. Line 1 defines the number of processes to run (we will vary this to see the effect on state space), line 3 defines the counter, and line 4 is used to implement the assertion that appears on lines 29–39.

Lines 6–13 define a process that increments the counter non-atomically. The argument `me` is the process number, set by the initialization block later in the code. Because simple Promela statements are each assumed atomic, we must break the increment into the two statements on lines 10–11. The assignment on line 12 marks the process's completion. Because the Spin system will fully search the state space, including all possible sequences of states, there is no need for the loop that would be used for conventional stress testing.

Lines 15–40 are the initialization block, which is executed first. Lines 19–28 actually do the initialization, while lines 29–39 perform the assertion. Both are atomic blocks in order to avoid unnecessarily increasing the state space: because they are not part of the algorithm proper, we lose no verification coverage by making them atomic.

The `do-od` construct on lines 21–27 implements a Promela loop, which can be thought of as a C `for (;;)` loop containing a `switch` statement that allows expressions in case labels. The condition blocks (prefixed by `::`) are scanned non-deterministically, though in this case only one of the conditions can possibly hold at a given time. The first block of the `do-od` from lines 22–25 initializes the i-th incrementer's progress cell, runs the i-th incrementer's process, and then increments the variable `i`. The second block of the `do-od` on line 26 exits the loop once these processes have been started.

The atomic block on lines 29–39 also contains a similar `do-od` loop that sums up the progress counters. The `assert()` statement on line 38 verifies that if all processes have been completed, then all counts have been correctly recorded.

You can build and run this program as follows:

```
spin -a increment.spin      # Translate the model to C
cc -DSAFETY -o pan pan.c    # Compile the model
./pan                       # Run the model
```

This will produce output as shown in Listing 12.2. The first line tells us that our assertion was violated (as expected given the non-atomic increment!). The second line that a `trail` file was written describing how the assertion was violated. The "Warning" line reiterates that all was not well with our model. The second paragraph describes the type of state-search being carried out, in this case for assertion violations and invalid end states. The third paragraph gives state-size statistics: this small model had only 45 states. The final line shows memory usage.

The `trail` file may be rendered human-readable as follows:

```
spin -t -p increment.spin
```

This gives the output shown in Listing 12.3. As can be seen, the first portion of the init block created both incrementer processes, both of which first fetched the

Listing 12.2: Non-Atomic Increment Spin Output

```
pan:1: assertion violated
       ((sum<2)||(counter==2)) (at depth 22)
pan: wrote increment.spin.trail

(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
       + Partial Order Reduction

Full statespace search for:
       never claim             - (none specified)
       assertion violations    +
       cycle checks            - (disabled by -DSAFETY)
       invalid end states      +

State-vector 48 byte, depth reached 24, errors: 1
       45 states, stored
       13 states, matched
       58 transitions (= stored+matched)
       53 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.003       equivalent memory usage for states
                  (stored*(State-vector + overhead))
    0.290       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage
```

Table 12.1: Memory Usage of Increment Model

# incrementers	# states	total memory usage (MB)
1	11	128.7
2	52	128.7
3	372	128.7
4	3,496	128.9
5	40,221	131.7
6	545,720	174.0
7	8,521,446	881.9

counter, then both incremented and stored it, losing a count. The assertion then triggered, after which the global state is displayed.

12.1.1.2 Warm-Up: Atomic Increment

It is easy to fix this example by placing the body of the incrementer processes in an atomic block as shown in Listing 12.4. One could also have simply replaced the pair of statements with `counter = counter + 1`, because Promela statements are atomic. Either way, running this modified model gives us an error-free traversal of the state space, as shown in Listing 12.5.

Table 12.1 shows the number of states and memory consumed as a function of number of incrementers modeled (by redefining `NUMPROCS`):

Running unnecessarily large models is thus subtly discouraged, although 882 MB is well within the limits of modern desktop and laptop machines.

With this example under our belt, let's take a closer look at the commands used to analyze Promela models and then look at more elaborate examples.

12.1.2 How to Use Promela

Given a source file `qrcu.spin`, one can use the following commands:

`spin -a qrcu.spin`
: Create a file `pan.c` that fully searches the state machine.

`cc -DSAFETY [-DCOLLAPSE] [-DMA=N] -o pan pan.c`
: Compile the generated state-machine search. The `-DSAFETY` generates optimizations that are appropriate if you have only assertions (and perhaps `never` statements). If you have liveness, fairness, or forward-progress checks, you may need to compile without `-DSAFETY`. If you leave off `-DSAFETY` when you could have used it, the program will let you know.

  The optimizations produced by `-DSAFETY` greatly speed things up, so you should use it when you can. An example situation where you cannot use `-DSAFETY` is when checking for livelocks (AKA "non-progress cycles") via `-DNP`.

  The optional `-DCOLLAPSE` generates code for a state vector compression mode.

  Another optional flag `-DMA=N` generates code for a slow but aggressive state-space memory compression mode.

`./pan [-mN] [-wN]`
: This actually searches the state space. The number of states can reach into the tens of millions with very small state machines, so you will need a machine with large memory. For example, `qrcu.spin` with 3 updaters and 2 readers required 10.5 GB of memory even with the `-DCOLLAPSE` flag.

  If you see a message from `./pan` saying: "error: max search depth too small", you need to increase the maximum depth by a `-mN` option for a complete search. The default is `-m10000`.

www.dbooks.org

Listing 12.3: Non-Atomic Increment Error Trail

```
using statement merging
  1:    proc  0 (:init::1) increment.spin:21 (state 1)  [i = 0]
  2:    proc  0 (:init::1) increment.spin:23 (state 2)  [((i<2))]
  2:    proc  0 (:init::1) increment.spin:24 (state 3)  [progress[i] = 0]
Starting incrementer with pid 1
  3:    proc  0 (:init::1) increment.spin:25 (state 4)  [(run incrementer(i))]
  4:    proc  0 (:init::1) increment.spin:26 (state 5)  [i = (i+1)]
  5:    proc  0 (:init::1) increment.spin:23 (state 2)  [((i<2))]
  5:    proc  0 (:init::1) increment.spin:24 (state 3)  [progress[i] = 0]
Starting incrementer with pid 2
  6:    proc  0 (:init::1) increment.spin:25 (state 4)  [(run incrementer(i))]
  7:    proc  0 (:init::1) increment.spin:26 (state 5)  [i = (i+1)]
  8:    proc  0 (:init::1) increment.spin:27 (state 6)  [((i>=2))]
  9:    proc  0 (:init::1) increment.spin:22 (state 10) [break]
 10:    proc  2 (incrementer:1) increment.spin:11 (state 1)     [temp = counter]
 11:    proc  1 (incrementer:1) increment.spin:11 (state 1)     [temp = counter]
 12:    proc  2 (incrementer:1) increment.spin:12 (state 2)     [counter = (temp+1)]
 13:    proc  2 (incrementer:1) increment.spin:13 (state 3)     [progress[me] = 1]
 14: proc 2 terminates
 15:    proc  1 (incrementer:1) increment.spin:12 (state 2)     [counter = (temp+1)]
 16:    proc  1 (incrementer:1) increment.spin:13 (state 3)     [progress[me] = 1]
 17: proc 1 terminates
 18:    proc  0 (:init::1) increment.spin:31 (state 12) [i = 0]
 18:    proc  0 (:init::1) increment.spin:32 (state 13) [sum = 0]
 19:    proc  0 (:init::1) increment.spin:34 (state 14) [((i<2))]
 19:    proc  0 (:init::1) increment.spin:35 (state 15) [sum = (sum+progress[i])]
 19:    proc  0 (:init::1) increment.spin:36 (state 16) [i = (i+1)]
 20:    proc  0 (:init::1) increment.spin:34 (state 14) [((i<2))]
 20:    proc  0 (:init::1) increment.spin:35 (state 15) [sum = (sum+progress[i])]
 20:    proc  0 (:init::1) increment.spin:36 (state 16) [i = (i+1)]
 21:    proc  0 (:init::1) increment.spin:37 (state 17) [((i>=2))]
 22:    proc  0 (:init::1) increment.spin:33 (state 21) [break]
spin: increment.spin:39, Error: assertion violated
spin: text of failed assertion: assert(((sum<2)||(counter==2)))
 23:    proc  0 (:init::1) increment.spin:39 (state 22) [assert(((sum<2)||(counter==2)))]
spin: trail ends after 23 steps
#processes: 1
                counter = 1
                progress[0] = 1
                progress[1] = 1
 23:    proc  0 (:init::1) increment.spin:41 (state 24) <valid end state>
3 processes created
```

Listing 12.4: Promela Code for Atomic Increment

```
1 proctype incrementer(byte me)
2 {
3   int temp;
4
5   atomic {
6     temp = counter;
7     counter = temp + 1;
8   }
9   progress[me] = 1;
10 }
```

Listing 12.5: Atomic Increment Spin Output

```
(Spin Version 6.4.8 -- 2 March 2018)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 48 byte, depth reached 22, errors: 0
       52 states, stored
       21 states, matched
       73 transitions (= stored+matched)
       68 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.004       equivalent memory usage for states
                  (stored*(State-vector + overhead))
    0.290       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage

unreached in proctype incrementer
        (0 of 5 states)
unreached in init
        (0 of 24 states)
```

The `-wN` option specifies the hashtable size. The default for full state-space search is `-w24`.[1]

If you aren't sure whether your machine has enough memory, run `top` in one window and `./pan` in another. Keep the focus on the `./pan` window so that you can quickly kill execution if need be. As soon as CPU time drops much below 100 %, kill `./pan`. If you have removed focus from the window running `./pan`, you may wait a long time for the windowing system to grab enough memory to do anything for you.

Another option to avoid memory exhaustion is the `-DMEMLIM=N` compiler flag. `-DMEMLIM=2000` would set the maximum of 2 GB.

Don't forget to capture the output, especially if you are working on a remote machine.

If your model includes forward-progress checks, you will likely need to enable "weak fairness" via the `-f` command-line argument to `./pan`. If your forward-progress checks involve `accept` labels, you will also need the `-a` argument.

`spin -t -p qrcu.spin`
: Given `trail` file output by a run that encountered an error, output the sequence of steps leading to that error. The `-g` flag will also include the values of changed global variables, and the `-l` flag will also include the values of changed local variables.

12.1.2.1 Promela Peculiarities

Although all computer languages have underlying similarities, Promela will provide some surprises to people used to coding in C, C++, or Java.

1. In C, "`;`" terminates statements. In Promela it separates them. Fortunately, more recent versions of Spin have become much more forgiving of "extra" semicolons.

2. Promela's looping construct, the `do` statement, takes conditions. This `do` statement closely resembles a looping if-then-else statement.

3. In C's `switch` statement, if there is no matching case, the whole statement is skipped. In Promela's equivalent, confusingly called `if`, if there is no matching guard expression, you get an error without a recognizable corresponding error message. So, if the error output indicates an innocent line of code, check to see if you left out a condition from an `if` or `do` statement.

4. When creating stress tests in C, one usually races suspect operations against each other repeatedly. In Promela, one instead sets up a single race, because Promela will search out all the possible outcomes from that single race. Sometimes you do need to loop in Promela, for example, if multiple operations overlap, but doing so greatly increases the size of your state space.

5. In C, the easiest thing to do is to maintain a loop counter to track progress and terminate the loop. In Promela, loop counters must be avoided like the

[1] As of Spin Version 6.4.6 and 6.4.8. In the online manual of Spin dated 10 July 2011, the default for exhaustive search mode is said to be `-w19`, which does not meet the actual behavior.

www.dbooks.org

plague because they cause the state space to explode. On the other hand, there is no penalty for infinite loops in Promela as long as none of the variables monotonically increase or decrease—Promela will figure out how many passes through the loop really matter, and automatically prune execution beyond that point.

6. In C torture-test code, it is often wise to keep per-task control variables. They are cheap to read, and greatly aid in debugging the test code. In Promela, per-task control variables should be used only when there is no other alternative. To see this, consider a 5-task verification with one bit each to indicate completion. This gives 32 states. In contrast, a simple counter would have only six states, more than a five-fold reduction. That factor of five might not seem like a problem, at least not until you are struggling with a verification program possessing more than 150 million states consuming more than 10 GB of memory!

7. One of the most challenging things both in C torture-test code and in Promela is formulating good assertions. Promela also allows `never` claims that act like an assertion replicated between every line of code.

8. Dividing and conquering is extremely helpful in Promela in keeping the state space under control. Splitting a large model into two roughly equal halves will result in the state space of each half being roughly the square root of the whole. For example, a million-state combined model might reduce to a pair of thousand-state models. Not only will Promela handle the two smaller models much more quickly with much less memory, but the two smaller algorithms are easier for people to understand.

12.1.2.2 Promela Coding Tricks

Promela was designed to analyze protocols, so using it on parallel programs is a bit abusive. The following tricks can help you to abuse Promela safely:

1. Memory reordering. Suppose you have a pair of statements copying globals x and y to locals r1 and r2, where ordering matters (e.g., unprotected by locks), but where you have no memory barriers. This can be modeled in Promela as follows:

Listing 12.6: Complex Promela Assertion

```
1 i = 0;
2 sum = 0;
3 do
4 :: i < N_QRCU_READERS ->
5   sum = sum + (readerstart[i] == 1 &&
6                readerprogress[i] == 1);
7   i++
8 :: i >= N_QRCU_READERS ->
9   assert(sum == 0);
10  break
11 od
```

```
1 if
2 :: 1 -> r1 = x;
3         r2 = y
4 :: 1 -> r2 = y;
5         r1 = x
6 fi
```

   The two branches of the `if` statement will be selected nondeterministically, since they both are available. Because the full state space is searched, *both* choices will eventually be made in all cases.

   Of course, this trick will cause your state space to explode if used too heavily. In addition, it requires you to anticipate possible reorderings.

2. State reduction. If you have complex assertions, evaluate them under `atomic`. After all, they are not part of the algorithm. One example of a complex assertion (to be discussed in more detail later) is as shown in Listing 12.6.

   There is no reason to evaluate this assertion non-atomically, since it is not actually part of the algorithm. Because each statement contributes to state, we can reduce the number of useless states by enclosing it in an `atomic` block as shown in Listing 12.7.

3. Promela does not provide functions. You must instead use C preprocessor macros. However, you must use them carefully in order to avoid combinatorial explosion.

Now we are ready for further examples.

12.1.3 Promela Example: Locking

Since locks are generally useful, `spin_lock()` and `spin_unlock()` macros are provided in `lock.h`, which may be included from multiple Promela models, as shown in Listing 12.8. The `spin_lock()` macro contains an infinite `do-od` loop spanning lines 2–11, courtesy of the

Listing 12.7: Atomic Block for Complex Promela Assertion

```
1 atomic {
2   i = 0;
3   sum = 0;
4   do
5   :: i < N_QRCU_READERS ->
6     sum = sum + (readerstart[i] == 1 &&
7                  readerprogress[i] == 1);
8     i++
9   :: i >= N_QRCU_READERS ->
10    assert(sum == 0);
11    break
12  od
13 }
```

Listing 12.8: Promela Code for Spinlock

```
1 #define spin_lock(mutex) \
2   do \
3   :: 1 -> atomic { \
4       if \
5       :: mutex == 0 -> \
6         mutex = 1; \
7         break \
8       :: else -> skip \
9       fi \
10    } \
11  od
12
13 #define spin_unlock(mutex) \
14   mutex = 0
```

single guard expression of "1" on line 3. The body of this loop is a single atomic block that contains an `if-fi` statement. The `if-fi` construct is similar to the `do-od` construct, except that it takes a single pass rather than looping. If the lock is not held on line 5, then line 6 acquires it and line 7 breaks out of the enclosing `do-od` loop (and also exits the atomic block). On the other hand, if the lock is already held on line 8, we do nothing (`skip`), and fall out of the `if-fi` and the atomic block so as to take another pass through the outer loop, repeating until the lock is available.

The `spin_unlock()` macro simply marks the lock as no longer held.

Note that memory barriers are not needed because Promela assumes full ordering. In any given Promela state, all processes agree on both the current state and the order of state changes that caused us to arrive at the current state. This is analogous to the "sequentially consistent" memory model used by a few computer systems (such as 1990s MIPS and PA-RISC). As noted earlier, and as will be seen in a later example, weak memory ordering must be explicitly coded.

These macros are tested by the Promela code shown in Listing 12.9. This code is similar to that used to test the increments, with the number of locking processes defined by the `N_LOCKERS` macro definition on line 3. The mutex

Listing 12.9: Promela Code to Test Spinlocks

```
1 #include "lock.h"
2
3 #define N_LOCKERS 3
4
5 bit mutex = 0;
6 bit havelock[N_LOCKERS];
7 int sum;
8
9 proctype locker(byte me)
10 {
11   do
12   :: 1 ->
13     spin_lock(mutex);
14     havelock[me] = 1;
15     havelock[me] = 0;
16     spin_unlock(mutex)
17   od
18 }
19
20 init {
21   int i = 0;
22   int j;
23
24 end:  do
25   :: i < N_LOCKERS ->
26     havelock[i] = 0;
27     run locker(i);
28     i++
29   :: i >= N_LOCKERS ->
30     sum = 0;
31     j = 0;
32     atomic {
33       do
34       :: j < N_LOCKERS ->
35         sum = sum + havelock[j];
36         j = j + 1
37       :: j >= N_LOCKERS ->
38         break
39       od
40     }
41     assert(sum <= 1);
42     break
43   od
44 }
```

itself is defined on line 5, an array to track the lock owner on line 6, and line 7 is used by assertion code to verify that only one process holds the lock.

The locker process is on lines 9–18, and simply loops forever acquiring the lock on line 13, claiming it on line 14, unclaiming it on line 15, and releasing it on line 16.

The init block on lines 20–44 initializes the current locker's havelock array entry on line 26, starts the current locker on line 27, and advances to the next locker on line 28. Once all locker processes are spawned, the `do-od` loop moves to line 29, which checks the assertion. Lines 30 and 31 initialize the control variables, lines 32–40 atomically sum the havelock array entries, line 41 is the assertion, and line 42 exits the loop.

We can run this model by placing the two code fragments of Listings 12.8 and 12.9 into files named `lock.h` and

www.dbooks.org

lock.spin, respectively, and then running the following commands:

```
spin -a lock.spin
cc -DSAFETY -o pan pan.c
./pan
```

Listing 12.10: Output for Spinlock Test

```
(Spin Version 6.4.8 -- 2 March 2018)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 52 byte, depth reached 360, errors: 0
      576 states, stored
      929 states, matched
     1505 transitions (= stored+matched)
      368 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.044       equivalent memory usage for states
                  (stored*(State-vector + overhead))
    0.288       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  128.730       total actual memory usage

unreached in proctype locker
        lock.spin:19, state 20, "-end-"
        (1 of 20 states)
unreached in init
        (0 of 22 states)
```

The output will look something like that shown in Listing 12.10. As expected, this run has no assertion failures ("errors: 0").

Quick Quiz 12.1: Why is there an unreached statement in locker? After all, isn't this a *full* state-space search? ■

Quick Quiz 12.2: What are some Promela code-style issues with this example? ■

12.1.4 Promela Example: QRCU

This final example demonstrates a real-world use of Promela on Oleg Nesterov's QRCU [Nes06a, Nes06b], but modified to speed up the synchronize_qrcu() fastpath.

But first, what is QRCU?

QRCU is a variant of SRCU [McK06] that trades somewhat higher read overhead (atomic increment and decrement on a global variable) for extremely low grace-period latencies. If there are no readers, the grace period will be detected in less than a microsecond, compared to the multi-millisecond grace-period latencies of most other RCU implementations.

1. There is a qrcu_struct that defines a QRCU domain. Like SRCU (and unlike other variants of RCU) QRCU's action is not global, but instead focused on the specified qrcu_struct.

2. There are qrcu_read_lock() and qrcu_read_unlock() primitives that delimit QRCU read-side critical sections. The corresponding qrcu_struct must be passed into these primitives, and the return value from qrcu_read_lock() must be passed to qrcu_read_unlock().

   For example:

   ```
   idx = qrcu_read_lock(&my_qrcu_struct);
   /* read-side critical section. */
   qrcu_read_unlock(&my_qrcu_struct, idx);
   ```

3. There is a synchronize_qrcu() primitive that blocks until all pre-existing QRCU read-side critical sections complete, but, like SRCU's synchronize_srcu(), QRCU's synchronize_qrcu() need wait only for those read-side critical sections that are using the same qrcu_struct.

   For example, synchronize_qrcu(&your_qrcu_struct) would *not* need to wait on the earlier QRCU read-side critical section. In contrast, synchronize_qrcu(&my_qrcu_struct) *would* need to wait, since it shares the same qrcu_struct.

A Linux-kernel patch for QRCU has been produced [McK07c], but is unlikely to ever be included in the Linux kernel.

Listing 12.11: QRCU Global Variables

```
1 #include "lock.h"
2
3 #define N_QRCU_READERS 2
4 #define N_QRCU_UPDATERS 2
5
6 bit idx = 0;
7 byte ctr[2];
8 byte readerprogress[N_QRCU_READERS];
9 bit mutex = 0;
```

Returning to the Promela code for QRCU, the global variables are as shown in Listing 12.11. This example uses locking and includes lock.h. Both the number of readers and writers can be varied using the two #define statements, giving us not one but two ways to create combinatorial explosion. The idx variable controls which

of the two elements of the ctr array will be used by readers, and the readerprogress variable allows an assertion to determine when all the readers are finished (since a QRCU update cannot be permitted to complete until all pre-existing readers have completed their QRCU read-side critical sections). The readerprogress array elements have values as follows, indicating the state of the corresponding reader:

0: not yet started.

1: within QRCU read-side critical section.

2: finished with QRCU read-side critical section.

Finally, the mutex variable is used to serialize updaters' slowpaths.

Listing 12.12: QRCU Reader Process

```
1  proctype qrcu_reader(byte me)
2  {
3    int myidx;
4
5    do
6    :: 1 ->
7      myidx = idx;
8      atomic {
9        if
10       :: ctr[myidx] > 0 ->
11         ctr[myidx]++;
12         break
13       :: else -> skip
14       fi
15     }
16   od;
17   readerprogress[me] = 1;
18   readerprogress[me] = 2;
19   atomic { ctr[myidx]-- }
20 }
```

QRCU readers are modeled by the qrcu_reader() process shown in Listing 12.12. A do-od loop spans lines 5–16, with a single guard of "1" on line 6 that makes it an infinite loop. Line 7 captures the current value of the global index, and lines 8–15 atomically increment it (and break from the infinite loop) if its value was non-zero (atomic_inc_not_zero()). Line 17 marks entry into the RCU read-side critical section, and line 18 marks exit from this critical section, both lines for the benefit of the assert() statement that we shall encounter later. Line 19 atomically decrements the same counter that we incremented, thereby exiting the RCU read-side critical section.

The C-preprocessor macro shown in Listing 12.13 sums the pair of counters so as to emulate weak memory ordering. Lines 2–13 fetch one of the counters, and line 14 fetches the other of the pair and sums them. The atomic block consists of a single do-od statement. This

Listing 12.13: QRCU Unordered Summation

```
1  #define sum_unordered \
2    atomic { \
3      do \
4      :: 1 -> \
5        sum = ctr[0]; \
6        i = 1; \
7        break \
8      :: 1 -> \
9        sum = ctr[1]; \
10       i = 0; \
11       break \
12     od; \
13   } \
14   sum = sum + ctr[i]
```

do-od statement (spanning lines 3–12) is unusual in that it contains two unconditional branches with guards on lines 4 and 8, which causes Promela to non-deterministically choose one of the two (but again, the full state-space search causes Promela to eventually make all possible choices in each applicable situation). The first branch fetches the zero-th counter and sets i to 1 (so that line 14 will fetch the first counter), while the second branch does the opposite, fetching the first counter and setting i to 0 (so that line 14 will fetch the second counter).

Quick Quiz 12.3: Is there a more straightforward way to code the do-od statement? ■

With the sum_unordered macro in place, we can now proceed to the update-side process shown in Listing 12.14. The update-side process repeats indefinitely, with the corresponding do-od loop ranging over lines 7–57. Each pass through the loop first snapshots the global readerprogress array into the local readerstart array on lines 12–21. This snapshot will be used for the assertion on line 53. Line 23 invokes sum_unordered, and then lines 24–27 re-invoke sum_unordered if the fastpath is potentially usable.

Lines 28–40 execute the slowpath code if need be, with lines 30 and 38 acquiring and releasing the update-side lock, lines 31–33 flipping the index, and lines 34–37 waiting for all pre-existing readers to complete.

Lines 44–56 then compare the current values in the readerprogress array to those collected in the readerstart array, forcing an assertion failure should any readers that started before this update still be in progress.

Quick Quiz 12.4: Why are there atomic blocks at lines 12–21 and lines 44–56, when the operations within those atomic blocks have no atomic implementation on any current production microprocessor? ■

www.dbooks.org

Listing 12.14: QRCU Updater Process

```
1  proctype qrcu_updater(byte me)
2  {
3    int i;
4    byte readerstart[N_QRCU_READERS];
5    int sum;
6
7    do
8    :: 1 ->
9
10     /* Snapshot reader state. */
11
12     atomic {
13       i = 0;
14       do
15       :: i < N_QRCU_READERS ->
16         readerstart[i] = readerprogress[i];
17         i++
18       :: i >= N_QRCU_READERS ->
19         break
20       od
21     }
22
23     sum_unordered;
24     if
25     :: sum <= 1 -> sum_unordered
26     :: else -> skip
27     fi;
28     if
29     :: sum > 1 ->
30       spin_lock(mutex);
31       atomic { ctr[!idx]++ }
32       idx = !idx;
33       atomic { ctr[!idx]-- }
34       do
35       :: ctr[!idx] > 0 -> skip
36       :: ctr[!idx] == 0 -> break
37       od;
38       spin_unlock(mutex);
39     :: else -> skip
40     fi;
41
42     /* Verify reader progress. */
43
44     atomic {
45       i = 0;
46       sum = 0;
47       do
48       :: i < N_QRCU_READERS ->
49         sum = sum + (readerstart[i] == 1 &&
50                      readerprogress[i] == 1);
51         i++
52       :: i >= N_QRCU_READERS ->
53         assert(sum == 0);
54         break
55       od
56     }
57   od
58 }
```

Quick Quiz 12.5: Is the re-summing of the counters on lines 24–27 *really* necessary? ■

Listing 12.15: QRCU Initialization Process

```
1  init {
2    int i;
3
4    atomic {
5      ctr[idx] = 1;
6      ctr[!idx] = 0;
7      i = 0;
8      do
9      :: i < N_QRCU_READERS ->
10       readerprogress[i] = 0;
11       run qrcu_reader(i);
12       i++
13     :: i >= N_QRCU_READERS -> break
14     od;
15     i = 0;
16     do
17     :: i < N_QRCU_UPDATERS ->
18       run qrcu_updater(i);
19       i++
20     :: i >= N_QRCU_UPDATERS -> break
21     od
22   }
23 }
```

All that remains is the initialization block shown in Listing 12.15. This block simply initializes the counter pair on lines 5–6, spawns the reader processes on lines 7–14, and spawns the updater processes on lines 15–21. This is all done within an atomic block to reduce state space.

12.1.4.1 Running the QRCU Example

To run the QRCU example, combine the code fragments in the previous section into a single file named `qrcu.spin`, and place the definitions for `spin_lock()` and `spin_unlock()` into a file named `lock.h`. Then use the following commands to build and run the QRCU model:

```
spin -a qrcu.spin
cc -DSAFETY [-DCOLLAPSE] -o pan pan.c
./pan [-mN]
```

The output shows that this model passes all of the cases shown in Table 12.2. It would be nice to run three readers and three updaters, however, simple extrapolation indicates that this will require about half a terabyte of memory. What to do?

It turns out that `./pan` gives advice when it runs out of memory, for example, when attempting to run three readers and three updaters:

```
hint: to reduce memory, recompile with
  -DCOLLAPSE # good, fast compression, or
  -DMA=96   # better/slower compression, or
  -DHC # hash-compaction, approximation
  -DBITSTATE # supertrace, approximation
```

Table 12.2: Memory Usage of QRCU Model

updaters	readers	# states	depth	memory (MB)[a]
1	1	376	95	128.7
1	2	6,177	218	128.9
1	3	99,728	385	132.6
2	1	29,399	859	129.8
2	2	1,071,181	2,352	169.6
2	3	33,866,736	12,857	1,540.8
3	1	2,749,453	53,809	236.6
3	2	186,202,860	328,014	10,483.7

[a] Obtained with the compiler flag `-DCOLLAPSE` specified.

Let's try the suggested compiler flag `-DMA=N`, which generates code for aggressive compression of the state space at the cost of greatly increased search overhead. The required commands are as follows:

```
spin -a qrcu.spin
cc -DSAFETY -DMA=96 -O2 -o pan pan.c
./pan -m20000000
```

Here, the depth limit of 20,000,000 is an order of magnitude larger than the expected depth deduced from simple extrapolation. Although this increases up-front memory usage, it avoids wasting a long run due to incomplete search resulting from a too-tight depth limit. This run took a little more than 3 days on a POWER9 server. The result is shown in Listing 12.16. This Spin run completed successfully with a total memory usage of only 6.5 GB, which is almost two orders of magnitude lower than the `-DCOLLAPSE` usage of about half a terabyte.

Quick Quiz 12.6: A compression rate of 0.48 % corresponds to a 200-to-1 decrease in memory occupied by the states! Is the state-space search *really* exhaustive??? ■

For reference, Table 12.3 summarizes the Spin results with `-DCOLLAPSE` and `-DMA=N` compiler flags. The memory usage is obtained with minimal sufficient search depths and `-DMA=N` parameters shown in the table. Hashtable sizes for `-DCOLLAPSE` runs are tweaked by the `-wN` option of `./pan` to avoid using too much memory hashing small state spaces. Hence the memory usage is smaller than what is shown in Table 12.2, where the hashtable size starts from the default of `-w24`. The runtime is from a POWER9 server, which shows that `-DMA=N` suffers up to about an order of magnitude higher CPU overhead than does `-DCOLLAPSE`, but on the other hand reduces memory overhead by well over an order of magnitude.

Listing 12.16: 3 Readers 3 Updaters QRCU Spin Output with `-DMA=96`

```
(Spin Version 6.4.6 -- 2 December 2016)
        + Partial Order Reduction
        + Graph Encoding (-DMA=96)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 96 byte, depth reached 2055621, errors: 0
MA stats: -DMA=84 is sufficient
Minimized Automaton:    56420520 nodes and 1.75128e+08 edges
9.6647071e+09 states, stored
9.7503813e+09 states, matched
1.9415088e+10 transitions (= stored+matched)
7.2047951e+09 atomic steps

Stats on memory usage (in Megabytes):
1142905.887     equivalent memory usage for states
                  (stored*(State-vector + overhead))
 5448.879       actual memory usage for states
                  (compression: 0.48%)
 1068.115       memory used for DFS stack (-m20000000)
    1.619       memory lost to fragmentation
 6515.375       total actual memory usage

unreached in proctype qrcu_reader
        (0 of 18 states)
unreached in proctype qrcu_updater
        qrcu.spin:102, state 82, "-end-"
        (1 of 82 states)
unreached in init
        (0 of 23 states)

pan: elapsed time 2.72e+05 seconds
pan: rate 35500.523 states/second
```

So far so good. But adding a few more updaters or readers would exhaust memory, even with `-DMA=N`.[2] So what to do? Here are some possible approaches:

1. See whether a smaller number of readers and updaters suffice to prove the general case.
2. Manually construct a proof of correctness.
3. Use a more capable tool.
4. Divide and conquer.

The following sections discuss each of these approaches.

12.1.4.2 How Many Readers and Updaters Are Really Needed?

One approach is to look carefully at the Promela code for `qrcu_updater()` and notice that the only global state change is happening under the lock. Therefore, only one

[2] Alternatively, the CPU consumption would become excessive.

www.dbooks.org

Table 12.3: QRCU Spin Result Summary

				-DCOLLAPSE			-DMA=N		
updaters	readers	# states	depth reached	-wN	memory (MB)	runtime (s)	N	memory (MB)	runtime (s)
1	1	376	95	12	0.10	0.00	40	0.29	0.00
1	2	6,177	218	12	0.39	0.01	47	0.59	0.02
1	3	99,728	385	16	4.60	0.14	54	3.04	0.45
2	1	29,399	859	16	2.30	0.03	55	0.70	0.13
2	2	1,071,181	2,352	20	49.24	1.45	62	7.77	5.76
2	3	33,866,736	12,857	24	1,540.70	62.5	69	111.66	326
3	1	2,749,453	53,809	21	125.25	4.01	70	11.41	19.5
3	2	186,202,860	328,014	28	10,482.51	390	77	222.26	2,560
3	3	9,664,707,100	2,055,621				84	5,557.02	266,000

updater at a time can possibly be modifying state visible to either readers or other updaters. This means that any sequences of state changes can be carried out serially by a single updater due to the fact that Promela does a full state-space search. Therefore, at most two updaters are required: one to change state and a second to become confused.

The situation with the readers is less clear-cut, as each reader does only a single read-side critical section then terminates. It is possible to argue that the useful number of readers is limited, due to the fact that the fastpath must see at most a zero and a one in the counters. This is a fruitful avenue of investigation, in fact, it leads to the full proof of correctness described in the next section.

12.1.4.3 Alternative Approach: Proof of Correctness

An informal proof [McK07c] follows:

1. For `synchronize_qrcu()` to exit too early, then by definition there must have been at least one reader present during `synchronize_qrcu()`'s full execution.
2. The counter corresponding to this reader will have been at least 1 during this time interval.
3. The `synchronize_qrcu()` code forces at least one of the counters to be at least 1 at all times.
4. Therefore, at any given point in time, either one of the counters will be at least 2, or both of the counters will be at least one.
5. However, the `synchronize_qrcu()` fastpath code can read only one of the counters at a given time. It is therefore possible for the fastpath code to fetch the first counter while zero, but to race with a counter flip so that the second counter is seen as one.
6. There can be at most one reader persisting through such a race condition, as otherwise the sum would be two or greater, which would cause the updater to take the slowpath.
7. But if the race occurs on the fastpath's first read of the counters, and then again on its second read, there have to have been two counter flips.
8. Because a given updater flips the counter only once, and because the update-side lock prevents a pair of updaters from concurrently flipping the counters, the only way that the fastpath code can race with a flip twice is if the first updater completes.
9. But the first updater will not complete until after all pre-existing readers have completed.
10. Therefore, if the fastpath races with a counter flip twice in succession, all pre-existing readers must have completed, so that it is safe to take the fastpath.

Of course, not all parallel algorithms have such simple proofs. In such cases, it may be necessary to enlist more capable tools.

12.1.4.4 Alternative Approach: More Capable Tools

Although Promela and Spin are quite useful, much more capable tools are available, particularly for verifying hardware. This means that if it is possible to translate your algorithm to the hardware-design VHDL language, as it often will be for low-level parallel algorithms, then it is

possible to apply these tools to your code (for example, this was done for the first realtime RCU algorithm). However, such tools can be quite expensive.

Although the advent of commodity multiprocessing might eventually result in powerful free-software model-checkers featuring fancy state-space-reduction capabilities, this does not help much in the here and now.

As an aside, there are Spin features that support approximate searches that require fixed amounts of memory, however, I have never been able to bring myself to trust approximations when verifying parallel algorithms.

Another approach might be to divide and conquer.

12.1.4.5 Alternative Approach: Divide and Conquer

It is often possible to break down a larger parallel algorithm into smaller pieces, which can then be proven separately. For example, a 10-billion-state model might be broken into a pair of 100,000-state models. Taking this approach not only makes it easier for tools such as Promela to verify your algorithms, it can also make your algorithms easier to understand.

12.1.4.6 Is QRCU Really Correct?

Is QRCU really correct? We have a Promela-based mechanical proof and a by-hand proof that both say that it is. However, a recent paper by Alglave et al. [AKT13] says otherwise (see Section 5.1 of the paper at the bottom of page 12). Which is it?

It turns out that both are correct! When QRCU was added to a suite of formal-verification benchmarks, its memory barriers were omitted, thus resulting in a buggy version of QRCU. So the real news here is that a number of formal-verification tools incorrectly proved this buggy QRCU correct. And this is why formal-verification tools themselves should be tested using bug-injected versions of the code being verified. If a given tool cannot find the injected bugs, then that tool is clearly untrustworthy.

Quick Quiz 12.7: But different formal-verification tools are often designed to locate particular classes of bugs. For example, very few formal-verification tools will find an error in the specification. So isn't this "clearly untrustworthy" judgment a bit harsh? ■

Therefore, if you do intend to use QRCU, please take care. Its proofs of correctness might or might not themselves be correct. Which is one reason why formal verification is unlikely to completely replace testing, as Donald Knuth pointed out so long ago.

Quick Quiz 12.8: Given that we have two independent proofs of correctness for the QRCU algorithm described herein, and given that the proof of incorrectness covers what is known to be a different algorithm, why is there any room for doubt? ■

12.1.5 Promela Parable: dynticks and Preemptible RCU

In early 2008, a preemptible variant of RCU was accepted into mainline Linux in support of real-time workloads, a variant similar to the RCU implementations in the -rt patchset [Mol05] since August 2005. Preemptible RCU is needed for real-time workloads because older RCU implementations disable preemption across RCU read-side critical sections, resulting in excessive real-time latencies.

However, one disadvantage of the older -rt implementation was that each grace period requires work to be done on each CPU, even if that CPU is in a low-power "dynticks-idle" state, and thus incapable of executing RCU read-side critical sections. The idea behind the dynticks-idle state is that idle CPUs should be physically powered down in order to conserve energy. In short, preemptible RCU can disable a valuable energy-conservation feature of recent Linux kernels. Although Josh Triplett and Paul McKenney had discussed some approaches for allowing CPUs to remain in low-power state throughout an RCU grace period (thus preserving the Linux kernel's ability to conserve energy), matters did not come to a head until Steve Rostedt integrated a new dyntick implementation with preemptible RCU in the -rt patchset.

This combination caused one of Steve's systems to hang on boot, so in October, Paul coded up a dynticks-friendly modification to preemptible RCU's grace-period processing. Steve coded up `rcu_irq_enter()` and `rcu_irq_exit()` interfaces called from the `irq_enter()` and `irq_exit()` interrupt entry/exit functions. These `rcu_irq_enter()` and `rcu_irq_exit()` functions are needed to allow RCU to reliably handle situations where a dynticks-idle CPU is momentarily powered up for an interrupt handler containing RCU read-side critical sections. With these changes in place, Steve's system booted reliably, but Paul continued inspecting the code periodically on the assumption that we could not possibly have gotten the code right on the first try.

Paul reviewed the code repeatedly from October 2007 to February 2008, and almost always found at least one bug. In one case, Paul even coded and tested a fix before

www.dbooks.org

realizing that the bug was illusory, and in fact in all cases, the "bug" turned out to be illusory.

Near the end of February, Paul grew tired of this game. He therefore decided to enlist the aid of Promela and Spin. The following presents a series of seven increasingly realistic Promela models, the last of which passes, consuming about 40 GB of main memory for the state space.

More important, Promela and Spin did find a very subtle bug for me!

Quick Quiz 12.9: Yeah, that's just great! Now, just what am I supposed to do if I don't happen to have a machine with 40 GB of main memory??? ■

Still better would be to come up with a simpler and faster algorithm that has a smaller state space. Even better would be an algorithm so simple that its correctness was obvious to the casual observer!

Sections 12.1.5.1–12.1.5.4 give an overview of preemptible RCU's dynticks interface, followed by Section 12.1.6's discussion of the validation of the interface.

12.1.5.1 Introduction to Preemptible RCU and dynticks

The per-CPU `dynticks_progress_counter` variable is central to the interface between dynticks and preemptible RCU. This variable has an even value whenever the corresponding CPU is in dynticks-idle mode, and an odd value otherwise. A CPU exits dynticks-idle mode for the following three reasons:

1. To start running a task,
2. When entering the outermost of a possibly nested set of interrupt handlers, and
3. When entering an NMI handler.

Preemptible RCU's grace-period machinery samples the value of the `dynticks_progress_counter` variable in order to determine when a dynticks-idle CPU may safely be ignored.

The following three sections give an overview of the task interface, the interrupt/NMI interface, and the use of the `dynticks_progress_counter` variable by the grace-period machinery as of Linux kernel v2.6.25-rc4.

12.1.5.2 Task Interface

When a given CPU enters dynticks-idle mode because it has no more tasks to run, it invokes `rcu_enter_nohz()`:

```
1 static inline void rcu_enter_nohz(void)
2 {
3   mb();
4   __get_cpu_var(dynticks_progress_counter)++;
5   WARN_ON(__get_cpu_var(dynticks_progress_counter) &
6           0x1);
7 }
```

This function simply increments `dynticks_progress_counter` and checks that the result is even, but first executing a memory barrier to ensure that any other CPU that sees the new value of `dynticks_progress_counter` will also see the completion of any prior RCU read-side critical sections.

Similarly, when a CPU that is in dynticks-idle mode prepares to start executing a newly runnable task, it invokes `rcu_exit_nohz()`:

```
1 static inline void rcu_exit_nohz(void)
2 {
3   __get_cpu_var(dynticks_progress_counter)++;
4   mb();
5   WARN_ON(!(__get_cpu_var(dynticks_progress_counter) &
6             0x1));
7 }
```

This function again increments `dynticks_progress_counter`, but follows it with a memory barrier to ensure that if any other CPU sees the result of any subsequent RCU read-side critical section, then that other CPU will also see the incremented value of `dynticks_progress_counter`. Finally, `rcu_exit_nohz()` checks that the result of the increment is an odd value.

The `rcu_enter_nohz()` and `rcu_exit_nohz()` functions handle the case where a CPU enters and exits dynticks-idle mode due to task execution, but does not handle interrupts, which are covered in the following section.

12.1.5.3 Interrupt Interface

The `rcu_irq_enter()` and `rcu_irq_exit()` functions handle interrupt/NMI entry and exit, respectively. Of course, nested interrupts must also be properly accounted for. The possibility of nested interrupts is handled by a second per-CPU variable, `rcu_update_flag`, which is incremented upon entry to an interrupt or NMI handler (in `rcu_irq_enter()`) and is decremented upon exit (in `rcu_irq_exit()`). In addition, the pre-existing `in_interrupt()` primitive is used to distinguish between an outermost or a nested interrupt/NMI.

Interrupt entry is handled by the `rcu_irq_enter()` shown below:

```
1 void rcu_irq_enter(void)
2 {
```

```
3   int cpu = smp_processor_id();
4
5   if (per_cpu(rcu_update_flag, cpu))
6     per_cpu(rcu_update_flag, cpu)++;
7   if (!in_interrupt() &&
8       (per_cpu(dynticks_progress_counter,
9                cpu) & 0x1) == 0) {
10    per_cpu(dynticks_progress_counter, cpu)++;
11    smp_mb();
12    per_cpu(rcu_update_flag, cpu)++;
13  }
14 }
```

Line 3 fetches the current CPU's number, while lines 5 and 6 increment the `rcu_update_flag` nesting counter if it is already non-zero. Lines 7–9 check to see whether we are the outermost level of interrupt, and, if so, whether `dynticks_progress_counter` needs to be incremented. If so, line 10 increments `dynticks_progress_counter`, line 11 executes a memory barrier, and line 12 increments `rcu_update_flag`. As with `rcu_exit_nohz()`, the memory barrier ensures that any other CPU that sees the effects of an RCU read-side critical section in the interrupt handler (following the `rcu_irq_enter()` invocation) will also see the increment of `dynticks_progress_counter`.

Quick Quiz 12.10: Why not simply increment `rcu_update_flag`, and then only increment `dynticks_progress_counter` if the old value of `rcu_update_flag` was zero??? ■

Quick Quiz 12.11: But if line 7 finds that we are the outermost interrupt, wouldn't we *always* need to increment `dynticks_progress_counter`? ■

Interrupt exit is handled similarly by `rcu_irq_exit()`:

```
1  void rcu_irq_exit(void)
2  {
3    int cpu = smp_processor_id();
4
5    if (per_cpu(rcu_update_flag, cpu)) {
6      if (--per_cpu(rcu_update_flag, cpu))
7        return;
8      WARN_ON(in_interrupt());
9      smp_mb();
10     per_cpu(dynticks_progress_counter, cpu)++;
11     WARN_ON(per_cpu(dynticks_progress_counter,
12                     cpu) & 0x1);
13   }
14 }
```

Line 3 fetches the current CPU's number, as before. Line 5 checks to see if the `rcu_update_flag` is non-zero, returning immediately (via falling off the end of the function) if not. Otherwise, lines 6 through 12 come into play. Line 6 decrements `rcu_update_flag`, returning if the result is not zero. Line 8 verifies that we are indeed leaving the outermost level of nested interrupts, line 9

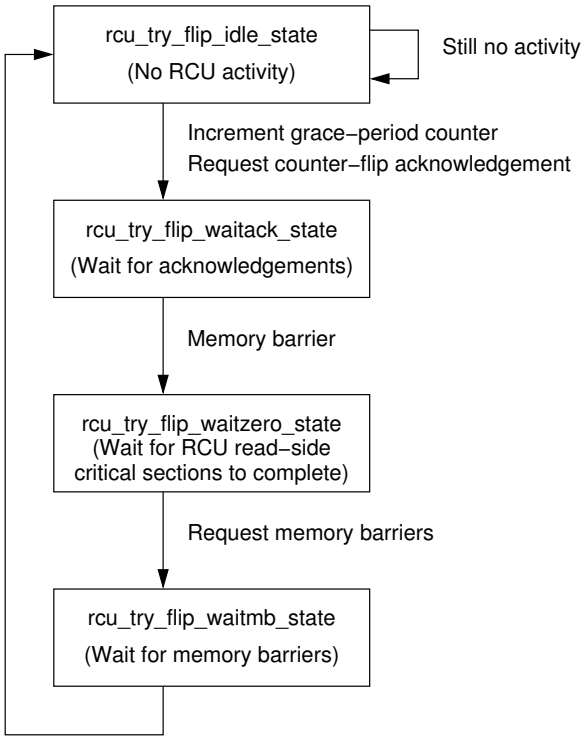


Figure 12.1: Preemptible RCU State Machine

executes a memory barrier, line 10 increments `dynticks_progress_counter`, and lines 11 and 12 verify that this variable is now even. As with `rcu_enter_nohz()`, the memory barrier ensures that any other CPU that sees the increment of `dynticks_progress_counter` will also see the effects of an RCU read-side critical section in the interrupt handler (preceding the `rcu_irq_exit()` invocation).

These two sections have described how the `dynticks_progress_counter` variable is maintained during entry to and exit from dynticks-idle mode, both by tasks and by interrupts and NMIs. The following section describes how this variable is used by preemptible RCU's grace-period machinery.

12.1.5.4 Grace-Period Interface

Of the four preemptible RCU grace-period states shown in Figure 12.1, only the `rcu_try_flip_waitack_state` and `rcu_try_flip_waitmb_state` states need to wait for other CPUs to respond.

Of course, if a given CPU is in dynticks-idle state, we shouldn't wait for it. Therefore, just before entering one of these two states, the preceding state takes a snapshot

www.dbooks.org

of each CPU's dynticks_progress_counter variable, placing the snapshot in another per-CPU variable, rcu_dyntick_snapshot. This is accomplished by invoking dyntick_save_progress_counter(), shown below:

```
1 static void dyntick_save_progress_counter(int cpu)
2 {
3   per_cpu(rcu_dyntick_snapshot, cpu) =
4     per_cpu(dynticks_progress_counter, cpu);
5 }
```

The rcu_try_flip_waitack_state state invokes rcu_try_flip_waitack_needed(), shown below:

```
1 static inline int
2 rcu_try_flip_waitack_needed(int cpu)
3 {
4   long curr;
5   long snap;
6
7   curr = per_cpu(dynticks_progress_counter, cpu);
8   snap = per_cpu(rcu_dyntick_snapshot, cpu);
9   smp_mb();
10  if ((curr == snap) && ((curr & 0x1) == 0))
11    return 0;
12  if ((curr - snap) > 2 || (snap & 0x1) == 0)
13    return 0;
14  return 1;
15 }
```

Lines 7 and 8 pick up current and snapshot versions of dynticks_progress_counter, respectively. The memory barrier on line 9 ensures that the counter checks in the later rcu_try_flip_waitzero_state follow the fetches of these counters. Lines 10 and 11 return zero (meaning no communication with the specified CPU is required) if that CPU has remained in dynticks-idle state since the time that the snapshot was taken. Similarly, lines 12 and 13 return zero if that CPU was initially in dynticks-idle state or if it has completely passed through a dynticks-idle state. In both these cases, there is no way that the CPU could have retained the old value of the grace-period counter. If neither of these conditions hold, line 14 returns one, meaning that the CPU needs to explicitly respond.

For its part, the rcu_try_flip_waitmb_state state invokes rcu_try_flip_waitmb_needed(), shown below:

```
1 static inline int
2 rcu_try_flip_waitmb_needed(int cpu)
3 {
4   long curr;
5   long snap;
6
7   curr = per_cpu(dynticks_progress_counter, cpu);
8   snap = per_cpu(rcu_dyntick_snapshot, cpu);
9   smp_mb();
10  if ((curr == snap) && ((curr & 0x1) == 0))
11    return 0;
12  if (curr != snap)
13    return 0;
14  return 1;
15 }
```

This is quite similar to rcu_try_flip_waitack_needed(), the difference being in lines 12 and 13, because any transition either to or from dynticks-idle state executes the memory barrier needed by the rcu_try_flip_waitmb_state state.

We now have seen all the code involved in the interface between RCU and the dynticks-idle state. The next section builds up the Promela model used to verify this code.

Quick Quiz 12.12: Can you spot any bugs in any of the code in this section? ■

12.1.6 Validating Preemptible RCU and dynticks

This section develops a Promela model for the interface between dynticks and RCU step by step, with each of Sections 12.1.6.1–12.1.6.7 illustrating one step, starting with the process-level code, adding assertions, interrupts, and finally NMIs.

Section 12.1.6.8 lists lessons (re)learned during this effort, and Sections 12.1.6.9–12.1.6.15 present a simpler solution to RCU's dynticks problem.

12.1.6.1 Basic Model

This section translates the process-level dynticks entry/exit code and the grace-period processing into Promela [Hol03]. We start with rcu_exit_nohz() and rcu_enter_nohz() from the 2.6.25-rc4 kernel, placing these in a single Promela process that models exiting and entering dynticks-idle mode in a loop as follows:

```
1 proctype dyntick_nohz()
2 {
3   byte tmp;
4   byte i = 0;
5
6   do
7   :: i >= MAX_DYNTICK_LOOP_NOHZ -> break;
8   :: i < MAX_DYNTICK_LOOP_NOHZ ->
9     tmp = dynticks_progress_counter;
10    atomic {
11      dynticks_progress_counter = tmp + 1;
12      assert((dynticks_progress_counter & 1) == 1);
13    }
14    tmp = dynticks_progress_counter;
15    atomic {
16      dynticks_progress_counter = tmp + 1;
17      assert((dynticks_progress_counter & 1) == 0);
18    }
19    i++;
20  od;
21 }
```

Lines 6 and 20 define a loop. Line 7 exits the loop once the loop counter i has exceeded the limit MAX_DYNTICK_LOOP_NOHZ. Line 8 tells the loop construct to

execute lines 9–19 for each pass through the loop. Because the conditionals on lines 7 and 8 are exclusive of each other, the normal Promela random selection of true conditions is disabled. Lines 9 and 11 model rcu_exit_nohz()'s non-atomic increment of dynticks_progress_counter, while line 12 models the WARN_ON(). The atomic construct simply reduces the Promela state space, given that the WARN_ON() is not strictly speaking part of the algorithm. Lines 14–18 similarly model the increment and WARN_ON() for rcu_enter_nohz(). Finally, line 19 increments the loop counter.

Each pass through the loop therefore models a CPU exiting dynticks-idle mode (for example, starting to execute a task), then re-entering dynticks-idle mode (for example, that same task blocking).

Quick Quiz 12.13: Why isn't the memory barrier in rcu_exit_nohz() and rcu_enter_nohz() modeled in Promela? ■

Quick Quiz 12.14: Isn't it a bit strange to model rcu_exit_nohz() followed by rcu_enter_nohz()? Wouldn't it be more natural to instead model entry before exit? ■

The next step is to model the interface to RCU's grace-period processing. For this, we need to model dyntick_save_progress_counter(), rcu_try_flip_waitack_needed(), rcu_try_flip_waitmb_needed(), as well as portions of rcu_try_flip_waitack() and rcu_try_flip_waitmb(), all from the 2.6.25-rc4 kernel. The following grace_period() Promela process models these functions as they would be invoked during a single pass through preemptible RCU's grace-period processing.

```
1  proctype grace_period()
2  {
3    byte curr;
4    byte snap;
5
6    atomic {
7      printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
8      snap = dynticks_progress_counter;
9    }
10   do
11   :: 1 ->
12     atomic {
13       curr = dynticks_progress_counter;
14       if
15       :: (curr == snap) && ((curr & 1) == 0) ->
16         break;
17       :: (curr - snap) > 2 || (snap & 1) == 0 ->
18         break;
19       :: 1 -> skip;
20       fi;
21     }
22   od;
23   snap = dynticks_progress_counter;
24   do
25   :: 1 ->
26     atomic {
27       curr = dynticks_progress_counter;
28       if
29       :: (curr == snap) && ((curr & 1) == 0) ->
30         break;
31       :: (curr != snap) ->
32         break;
33       :: 1 -> skip;
34       fi;
35     }
36   od;
37 }
```

Lines 6–9 print out the loop limit (but only into the .trail file in case of error) and models a line of code from rcu_try_flip_idle() and its call to dyntick_save_progress_counter(), which takes a snapshot of the current CPU's dynticks_progress_counter variable. These two lines are executed atomically to reduce state space.

Lines 10–22 model the relevant code in rcu_try_flip_waitack() and its call to rcu_try_flip_waitack_needed(). This loop is modeling the grace-period state machine waiting for a counter-flip acknowledgement from each CPU, but only that part that interacts with dynticks-idle CPUs.

Line 23 models a line from rcu_try_flip_waitzero() and its call to dyntick_save_progress_counter(), again taking a snapshot of the CPU's dynticks_progress_counter variable.

Finally, lines 24–36 model the relevant code in rcu_try_flip_waitack() and its call to rcu_try_flip_waitack_needed(). This loop is modeling the grace-period state-machine waiting for each CPU to execute a memory barrier, but again only that part that interacts with dynticks-idle CPUs.

Quick Quiz 12.15: Wait a minute! In the Linux kernel, both dynticks_progress_counter and rcu_dyntick_snapshot are per-CPU variables. So why are they instead being modeled as single global variables? ■

The resulting model (dynticRCU-base.spin), when run with the runspin.sh script, generates 691 states and passes without errors, which is not at all surprising given that it completely lacks the assertions that could find failures. The next section therefore adds safety assertions.

12.1.6.2 Validating Safety

A safe RCU implementation must never permit a grace period to complete before the completion of any RCU readers that started before the start of the grace period.

www.dbooks.org

This is modeled by a grace_period_state variable that can take on three states as follows:

```
1 #define GP_IDLE      0
2 #define GP_WAITING   1
3 #define GP_DONE      2
4 byte grace_period_state = GP_DONE;
```

The grace_period() process sets this variable as it progresses through the grace-period phases, as shown below:

```
1  proctype grace_period()
2  {
3    byte curr;
4    byte snap;
5
6    grace_period_state = GP_IDLE;
7    atomic {
8      printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
9      snap = dynticks_progress_counter;
10     grace_period_state = GP_WAITING;
11   }
12   do
13   :: 1 ->
14     atomic {
15       curr = dynticks_progress_counter;
16       if
17       :: (curr == snap) && ((curr & 1) == 0) ->
18         break;
19       :: (curr - snap) > 2 || (snap & 1) == 0 ->
20         break;
21       :: 1 -> skip;
22       fi;
23     }
24   od;
25   grace_period_state = GP_DONE;
26   grace_period_state = GP_IDLE;
27   atomic {
28     snap = dynticks_progress_counter;
29     grace_period_state = GP_WAITING;
30   }
31   do
32   :: 1 ->
33     atomic {
34       curr = dynticks_progress_counter;
35       if
36       :: (curr == snap) && ((curr & 1) == 0) ->
37         break;
38       :: (curr != snap) ->
39         break;
40       :: 1 -> skip;
41       fi;
42     }
43   od;
44   grace_period_state = GP_DONE;
45 }
```

Lines 6, 10, 25, 26, 29, and 44 update this variable (combining atomically with algorithmic operations where feasible) to allow the dyntick_nohz() process to verify the basic RCU safety property. The form of this verification is to assert that the value of the grace_period_state variable cannot jump from GP_IDLE to GP_DONE during a time period over which RCU readers could plausibly persist.

Quick Quiz 12.16: Given there are a pair of back-to-back changes to grace_period_state on lines 25 and 26, how can we be sure that line 25's changes won't be lost? ■

The dyntick_nohz() Promela process implements this verification as shown below:

```
1  proctype dyntick_nohz()
2  {
3    byte tmp;
4    byte i = 0;
5    bit old_gp_idle;
6
7    do
8    :: i >= MAX_DYNTICK_LOOP_NOHZ -> break;
9    :: i < MAX_DYNTICK_LOOP_NOHZ ->
10     tmp = dynticks_progress_counter;
11     atomic {
12       dynticks_progress_counter = tmp + 1;
13       old_gp_idle = (grace_period_state == GP_IDLE);
14       assert((dynticks_progress_counter & 1) == 1);
15     }
16     atomic {
17       tmp = dynticks_progress_counter;
18       assert(!old_gp_idle ||
19              grace_period_state != GP_DONE);
20     }
21     atomic {
22       dynticks_progress_counter = tmp + 1;
23       assert((dynticks_progress_counter & 1) == 0);
24     }
25     i++;
26   od;
27 }
```

Line 13 sets a new old_gp_idle flag if the value of the grace_period_state variable is GP_IDLE at the beginning of task execution, and the assertion at lines 18 and 19 fire if the grace_period_state variable has advanced to GP_DONE during task execution, which would be illegal given that a single RCU read-side critical section could span the entire intervening time period.

The resulting model (dyntickRCU-base-s.spin), when run with the runspin.sh script, generates 964 states and passes without errors, which is reassuring. That said, although safety is critically important, it is also quite important to avoid indefinitely stalling grace periods. The next section therefore covers verifying liveness.

12.1.6.3 Validating Liveness

Although liveness can be difficult to prove, there is a simple trick that applies here. The first step is to make dyntick_nohz() indicate that it is done via a dyntick_nohz_done variable, as shown on line 27 of the following:

```
1  proctype dyntick_nohz()
2  {
3    byte tmp;
4    byte i = 0;
5    bit old_gp_idle;
6
```

```
7   do
8   :: i >= MAX_DYNTICK_LOOP_NOHZ -> break;
9   :: i < MAX_DYNTICK_LOOP_NOHZ ->
10    tmp = dynticks_progress_counter;
11    atomic {
12      dynticks_progress_counter = tmp + 1;
13      old_gp_idle = (grace_period_state == GP_IDLE);
14      assert((dynticks_progress_counter & 1) == 1);
15    }
16    atomic {
17      tmp = dynticks_progress_counter;
18      assert(!old_gp_idle ||
19             grace_period_state != GP_DONE);
20    }
21    atomic {
22      dynticks_progress_counter = tmp + 1;
23      assert((dynticks_progress_counter & 1) == 0);
24    }
25    i++;
26  od;
27  dyntick_nohz_done = 1;
28 }
```

With this variable in place, we can add assertions to `grace_period()` to check for unnecessary blockage as follows:

```
1  proctype grace_period()
2  {
3    byte curr;
4    byte snap;
5    bit shouldexit;
6
7    grace_period_state = GP_IDLE;
8    atomic {
9      printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10     shouldexit = 0;
11     snap = dynticks_progress_counter;
12     grace_period_state = GP_WAITING;
13   }
14   do
15   :: 1 ->
16     atomic {
17       assert(!shouldexit);
18       shouldexit = dyntick_nohz_done;
19       curr = dynticks_progress_counter;
20       if
21       :: (curr == snap) && ((curr & 1) == 0) ->
22         break;
23       :: (curr - snap) > 2 || (snap & 1) == 0 ->
24         break;
25       :: else -> skip;
26       fi;
27     }
28   od;
29   grace_period_state = GP_DONE;
30   grace_period_state = GP_IDLE;
31   atomic {
32     shouldexit = 0;
33     snap = dynticks_progress_counter;
34     grace_period_state = GP_WAITING;
35   }
36   do
37   :: 1 ->
38     atomic {
39       assert(!shouldexit);
40       shouldexit = dyntick_nohz_done;
41       curr = dynticks_progress_counter;
42       if
43       :: (curr == snap) && ((curr & 1) == 0) ->
44         break;
45       :: (curr != snap) ->
46         break;
47       :: else -> skip;
48       fi;
49     }
50   od;
51   grace_period_state = GP_DONE;
52 }
```

We have added the `shouldexit` variable on line 5, which we initialize to zero on line 10. Line 17 asserts that `shouldexit` is not set, while line 18 sets `shouldexit` to the `dyntick_nohz_done` variable maintained by `dyntick_nohz()`. This assertion will therefore trigger if we attempt to take more than one pass through the wait-for-counter-flip-acknowledgement loop after `dyntick_nohz()` has completed execution. After all, if `dyntick_nohz()` is done, then there cannot be any more state changes to force us out of the loop, so going through twice in this state means an infinite loop, which in turn means no end to the grace period.

Lines 32, 39, and 40 operate in a similar manner for the second (memory-barrier) loop.

However, running this model (`dynticRCU-base-sl-busted.spin`) results in failure, as line 23 is checking that the wrong variable is even. Upon failure, `spin` writes out a "trail" file (`dynticRCU-base-sl-busted.spin.trail`), which records the sequence of states that lead to the failure. Use the "`spin -t -p -g -l dynticRCU-base-sl-busted.spin`" command to cause `spin` to retrace this sequence of states, printing the statements executed and the values of variables (`dynticRCU-base-sl-busted.spin.trail.txt`). Note that the line numbers do not match the listing above due to the fact that `spin` takes both functions in a single file. However, the line numbers *do* match the full model (`dynticRCU-base-sl-busted.spin`).

We see that the `dyntick_nohz()` process completed at step 34 (search for "34:"), but that the `grace_period()` process nonetheless failed to exit the loop. The value of `curr` is 6 (see step 35) and that the value of `snap` is 5 (see step 17). Therefore the first condition on line 21 above does not hold because "`curr != snap`", and the second condition on line 23 does not hold either because `snap` is odd and because `curr` is only one greater than `snap`.

So one of these two conditions has to be incorrect. Referring to the comment block in `rcu_try_flip_waitack_needed()` for the first condition:

> If the CPU remained in dynticks mode for the entire time and didn't take any interrupts, NMIs, SMIs, or whatever, then it cannot be in the middle of an `rcu_read_lock()`, so the next

www.dbooks.org

> rcu_read_lock() it executes must use the new value of the counter. So we can safely pretend that this CPU already acknowledged the counter.

The first condition does match this, because if "curr == snap" and if curr is even, then the corresponding CPU has been in dynticks-idle mode the entire time, as required. So let's look at the comment block for the second condition:

> If the CPU passed through or entered a dynticks idle phase with no active irq handlers, then, as above, we can safely pretend that this CPU already acknowledged the counter.

The first part of the condition is correct, because if curr and snap differ by two, there will be at least one even number in between, corresponding to having passed completely through a dynticks-idle phase. However, the second part of the condition corresponds to having *started* in dynticks-idle mode, not having *finished* in this mode. We therefore need to be testing curr rather than snap for being an even number.

The corrected C code is as follows:

```
1 static inline int
2 rcu_try_flip_waitack_needed(int cpu)
3 {
4   long curr;
5   long snap;
6
7   curr = per_cpu(dynticks_progress_counter, cpu);
8   snap = per_cpu(rcu_dyntick_snapshot, cpu);
9   smp_mb();
10  if ((curr == snap) && ((curr & 0x1) == 0))
11    return 0;
12  if ((curr - snap) > 2 || (curr & 0x1) == 0)
13    return 0;
14  return 1;
15 }
```

Lines 10–13 can now be combined and simplified, resulting in the following. A similar simplification can be applied to rcu_try_flip_waitmb_needed().

```
1 static inline int
2 rcu_try_flip_waitack_needed(int cpu)
3 {
4   long curr;
5   long snap;
6
7   curr = per_cpu(dynticks_progress_counter, cpu);
8   snap = per_cpu(rcu_dyntick_snapshot, cpu);
9   smp_mb();
10  if ((curr - snap) >= 2 || (curr & 0x1) == 0)
11    return 0;
12  return 1;
13 }
```

Making the corresponding correction in the model (dyntickRCU-base-sl.spin) results in a correct verification with 661 states that passes without errors. However, it is worth noting that the first version of the liveness verification failed to catch this bug, due to a bug in the liveness verification itself. This liveness-verification bug was located by inserting an infinite loop in the grace_period() process, and noting that the liveness-verification code failed to detect this problem!

We have now successfully verified both safety and liveness conditions, but only for processes running and blocking. We also need to handle interrupts, a task taken up in the next section.

12.1.6.4 Interrupts

There are a couple of ways to model interrupts in Promela:

1. using C-preprocessor tricks to insert the interrupt handler between each and every statement of the dynticks_nohz() process, or
2. modeling the interrupt handler with a separate process.

A bit of thought indicated that the second approach would have a smaller state space, though it requires that the interrupt handler somehow run atomically with respect to the dynticks_nohz() process, but not with respect to the grace_period() process.

Fortunately, it turns out that Promela permits you to branch out of atomic statements. This trick allows us to have the interrupt handler set a flag, and recode dynticks_nohz() to atomically check this flag and execute only when the flag is not set. This can be accomplished with a C-preprocessor macro that takes a label and a Promela statement as follows:

```
1 #define EXECUTE_MAINLINE(label, stmt) \
2 label: skip;  \
3     atomic { \
4       if \
5       :: in_dyntick_irq -> goto label; \
6       :: else -> stmt; \
7       fi; \
8     }
```

One might use this macro as follows:

```
EXECUTE_MAINLINE(stmt1,
                 tmp = dynticks_progress_counter)
```

Line 2 of the macro creates the specified statement label. Lines 3–8 are an atomic block that tests the in_dyntick_irq variable, and if this variable is set (indicating that the

interrupt handler is active), branches out of the atomic block back to the label. Otherwise, line 6 executes the specified statement. The overall effect is that mainline execution stalls any time an interrupt is active, as required.

12.1.6.5 Validating Interrupt Handlers

The first step is to convert `dyntick_nohz()` to `EXECUTE_MAINLINE()` form, as follows:

```
1  proctype dyntick_nohz()
2  {
3    byte tmp;
4    byte i = 0;
5    bit old_gp_idle;
6
7    do
8    :: i >= MAX_DYNTICK_LOOP_NOHZ -> break;
9    :: i < MAX_DYNTICK_LOOP_NOHZ ->
10     EXECUTE_MAINLINE(stmt1,
11       tmp = dynticks_progress_counter)
12     EXECUTE_MAINLINE(stmt2,
13       dynticks_progress_counter = tmp + 1;
14       old_gp_idle = (grace_period_state == GP_IDLE);
15       assert((dynticks_progress_counter & 1) == 1))
16     EXECUTE_MAINLINE(stmt3,
17       tmp = dynticks_progress_counter;
18       assert(!old_gp_idle ||
19              grace_period_state != GP_DONE))
20     EXECUTE_MAINLINE(stmt4,
21       dynticks_progress_counter = tmp + 1;
22       assert((dynticks_progress_counter & 1) == 0))
23     i++;
24   od;
25   dyntick_nohz_done = 1;
26 }
```

It is important to note that when a group of statements is passed to `EXECUTE_MAINLINE()`, as in lines 12–15, all statements in that group execute atomically.

Quick Quiz 12.17: But what would you do if you needed the statements in a single `EXECUTE_MAINLINE()` group to execute non-atomically? ■

Quick Quiz 12.18: But what if the `dynticks_nohz()` process had "if" or "do" statements with conditions, where the statement bodies of these constructs needed to execute non-atomically? ■

The next step is to write a `dyntick_irq()` process to model an interrupt handler:

```
1  proctype dyntick_irq()
2  {
3    byte tmp;
4    byte i = 0;
5    bit old_gp_idle;
6
7    do
8    :: i >= MAX_DYNTICK_LOOP_IRQ -> break;
9    :: i < MAX_DYNTICK_LOOP_IRQ ->
10     in_dyntick_irq = 1;
11     if
12     :: rcu_update_flag > 0 ->
13       tmp = rcu_update_flag;
14       rcu_update_flag = tmp + 1;
15     :: else -> skip;
16     fi;
17     if
18     :: !in_interrupt &&
19        (dynticks_progress_counter & 1) == 0 ->
20       tmp = dynticks_progress_counter;
21       dynticks_progress_counter = tmp + 1;
22       tmp = rcu_update_flag;
23       rcu_update_flag = tmp + 1;
24     :: else -> skip;
25     fi;
26     tmp = in_interrupt;
27     in_interrupt = tmp + 1;
28     old_gp_idle = (grace_period_state == GP_IDLE);
29     assert(!old_gp_idle ||
30            grace_period_state != GP_DONE);
31     tmp = in_interrupt;
32     in_interrupt = tmp - 1;
33     if
34     :: rcu_update_flag != 0 ->
35       tmp = rcu_update_flag;
36       rcu_update_flag = tmp - 1;
37       if
38       :: rcu_update_flag == 0 ->
39         tmp = dynticks_progress_counter;
40         dynticks_progress_counter = tmp + 1;
41       :: else -> skip;
42       fi;
43     :: else -> skip;
44     fi;
45     atomic {
46       in_dyntick_irq = 0;
47       i++;
48     }
49   od;
50   dyntick_irq_done = 1;
51 }
```

The loop from lines 7–49 models up to `MAX_DYNTICK_LOOP_IRQ` interrupts, with lines 8 and 9 forming the loop condition and line 47 incrementing the control variable. Line 10 tells `dyntick_nohz()` that an interrupt handler is running, and line 46 tells `dyntick_nohz()` that this handler has completed. Line 50 is used for liveness verification, just like the corresponding line of `dyntick_nohz()`.

Quick Quiz 12.19: Why are lines 46 and 47 (the "`in_dyntick_irq = 0;`" and the "`i++;`") executed atomically? ■

Lines 11–25 model `rcu_irq_enter()`, and lines 26 and 27 model the relevant snippet of `__irq_enter()`. Lines 28–30 verify safety in much the same manner as do the corresponding lines of `dynticks_nohz()`. Lines 31 and 32 model the relevant snippet of `__irq_exit()`, and finally lines 33–44 model `rcu_irq_exit()`.

Quick Quiz 12.20: What property of interrupts is this `dynticks_irq()` process unable to model? ■

www.dbooks.org

The grace_period() process then becomes as follows:

```
1  proctype grace_period()
2  {
3    byte curr;
4    byte snap;
5    bit shouldexit;
6
7    grace_period_state = GP_IDLE;
8    atomic {
9      printf("MDLN = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10     printf("MDLI = %d\n", MAX_DYNTICK_LOOP_IRQ);
11     shouldexit = 0;
12     snap = dynticks_progress_counter;
13     grace_period_state = GP_WAITING;
14   }
15   do
16   :: 1 ->
17     atomic {
18       assert(!shouldexit);
19       shouldexit = dyntick_nohz_done && dyntick_irq_done;
20       curr = dynticks_progress_counter;
21       if
22       :: (curr - snap) >= 2 || (curr & 1) == 0 ->
23         break;
24       :: else -> skip;
25       fi;
26     }
27   od;
28   grace_period_state = GP_DONE;
29   grace_period_state = GP_IDLE;
30   atomic {
31     shouldexit = 0;
32     snap = dynticks_progress_counter;
33     grace_period_state = GP_WAITING;
34   }
35   do
36   :: 1 ->
37     atomic {
38       assert(!shouldexit);
39       shouldexit = dyntick_nohz_done && dyntick_irq_done;
40       curr = dynticks_progress_counter;
41       if
42       :: (curr != snap) || ((curr & 1) == 0) ->
43         break;
44       :: else -> skip;
45       fi;
46     }
47   od;
48   grace_period_state = GP_DONE;
49 }
```

The implementation of grace_period() is very similar to the earlier one. The only changes are the addition of line 10 to add the new interrupt-count parameter, changes to lines 19 and 39 to add the new dyntick_irq_done variable to the liveness checks, and of course the optimizations on lines 22 and 42.

This model (dyntickRCU-irqnn-ssl.spin) results in a correct verification with roughly half a million states, passing without errors. However, this version of the model does not handle nested interrupts. This topic is taken up in the next section.

12.1.6.6 Validating Nested Interrupt Handlers

Nested interrupt handlers may be modeled by splitting the body of the loop in dyntick_irq() as follows:

```
1  proctype dyntick_irq()
2  {
3    byte tmp;
4    byte i = 0;
5    byte j = 0;
6    bit old_gp_idle;
7    bit outermost;
8
9    do
10   :: i >= MAX_DYNTICK_LOOP_IRQ &&
11      j >= MAX_DYNTICK_LOOP_IRQ -> break;
12   :: i < MAX_DYNTICK_LOOP_IRQ ->
13     atomic {
14       outermost = (in_dyntick_irq == 0);
15       in_dyntick_irq = 1;
16     }
17     if
18     :: rcu_update_flag > 0 ->
19       tmp = rcu_update_flag;
20       rcu_update_flag = tmp + 1;
21     :: else -> skip;
22     fi;
23     if
24     :: !in_interrupt &&
25        (dynticks_progress_counter & 1) == 0 ->
26       tmp = dynticks_progress_counter;
27       dynticks_progress_counter = tmp + 1;
28       tmp = rcu_update_flag;
29       rcu_update_flag = tmp + 1;
30     :: else -> skip;
31     fi;
32     tmp = in_interrupt;
33     in_interrupt = tmp + 1;
34     atomic {
35       if
36       :: outermost ->
37         old_gp_idle = (grace_period_state == GP_IDLE);
38       :: else -> skip;
39       fi;
40     }
41     i++;
42   :: j < i ->
43     atomic {
44       if
45       :: j + 1 == i ->
46         assert(!old_gp_idle ||
47                grace_period_state != GP_DONE);
48       :: else -> skip;
49       fi;
50     }
51     tmp = in_interrupt;
52     in_interrupt = tmp - 1;
53     if
54     :: rcu_update_flag != 0 ->
55       tmp = rcu_update_flag;
56       rcu_update_flag = tmp - 1;
57       if
58       :: rcu_update_flag == 0 ->
59         tmp = dynticks_progress_counter;
60         dynticks_progress_counter = tmp + 1;
61       :: else -> skip;
62       fi;
63     :: else -> skip;
64     fi;
65     atomic {
66       j++;
67       in_dyntick_irq = (i != j);
68     }
```

```
69    od;
70    dyntick_irq_done = 1;
71  }
```

This is similar to the earlier `dynticks_irq()` process. It adds a second counter variable `j` on line 5, so that `i` counts entries to interrupt handlers and `j` counts exits. The `outermost` variable on line 7 helps determine when the `grace_period_state` variable needs to be sampled for the safety checks. The loop-exit check on lines 10 and 11 is updated to require that the specified number of interrupt handlers are exited as well as entered, and the increment of `i` is moved to line 41, which is the end of the interrupt-entry model. Lines 13–16 set the `outermost` variable to indicate whether this is the outermost of a set of nested interrupts and to set the `in_dyntick_irq` variable that is used by the `dyntick_nohz()` process. Lines 34–40 capture the state of the `grace_period_state` variable, but only when in the outermost interrupt handler.

Line 42 has the do-loop conditional for interrupt-exit modeling: as long as we have exited fewer interrupts than we have entered, it is legal to exit another interrupt. Lines 43–50 check the safety criterion, but only if we are exiting from the outermost interrupt level. Finally, lines 65–68 increment the interrupt-exit count `j` and, if this is the outermost interrupt level, clears `in_dyntick_irq`.

This model (`dyntickRCU-irq-ssl.spin`) results in a correct verification with a bit more than half a million states, passing without errors. However, this version of the model does not handle NMIs, which are taken up in the next section.

12.1.6.7 Validating NMI Handlers

We take the same general approach for NMIs as we do for interrupts, keeping in mind that NMIs do not nest. This results in a `dyntick_nmi()` process as follows:

```
1  proctype dyntick_nmi()
2  {
3    byte tmp;
4    byte i = 0;
5    bit old_gp_idle;
6
7    do
8    :: i >= MAX_DYNTICK_LOOP_NMI -> break;
9    :: i < MAX_DYNTICK_LOOP_NMI ->
10     in_dyntick_nmi = 1;
11     if
12     :: rcu_update_flag > 0 ->
13       tmp = rcu_update_flag;
14       rcu_update_flag = tmp + 1;
15     :: else -> skip;
16     fi;
17     if
18     :: !in_interrupt &&
19        (dynticks_progress_counter & 1) == 0 ->
20       tmp = dynticks_progress_counter;
21       dynticks_progress_counter = tmp + 1;
22       tmp = rcu_update_flag;
23       rcu_update_flag = tmp + 1;
24     :: else -> skip;
25     fi;
26     tmp = in_interrupt;
27     in_interrupt = tmp + 1;
28     old_gp_idle = (grace_period_state == GP_IDLE);
29     assert(!old_gp_idle ||
30            grace_period_state != GP_DONE);
31     tmp = in_interrupt;
32     in_interrupt = tmp - 1;
33     if
34     :: rcu_update_flag != 0 ->
35       tmp = rcu_update_flag;
36       rcu_update_flag = tmp - 1;
37       if
38       :: rcu_update_flag == 0 ->
39         tmp = dynticks_progress_counter;
40         dynticks_progress_counter = tmp + 1;
41       :: else -> skip;
42       fi;
43     :: else -> skip;
44     fi;
45     atomic {
46       i++;
47       in_dyntick_nmi = 0;
48     }
49   od;
50   dyntick_nmi_done = 1;
51 }
```

Of course, the fact that we have NMIs requires adjustments in the other components. For example, the `EXECUTE_MAINLINE()` macro now needs to pay attention to the NMI handler (`in_dyntick_nmi`) as well as the interrupt handler (`in_dyntick_irq`) by checking the `dyntick_nmi_done` variable as follows:

```
1 #define EXECUTE_MAINLINE(label, stmt) \
2 label: skip; \
3     atomic { \
4       if \
5       :: in_dyntick_irq || \
6          in_dyntick_nmi -> goto label; \
7       :: else -> stmt; \
8       fi; \
9     }
```

We will also need to introduce an `EXECUTE_IRQ()` macro that checks `in_dyntick_nmi` in order to allow `dyntick_irq()` to exclude `dyntick_nmi()`:

```
1 #define EXECUTE_IRQ(label, stmt) \
2 label: skip; \
3     atomic { \
4       if \
5       :: in_dyntick_nmi -> goto label; \
6       :: else -> stmt; \
7       fi; \
8     }
```

It is further necessary to convert `dyntick_irq()` to `EXECUTE_IRQ()` as follows:

```
1 proctype dyntick_irq()
2 {
```


www.dbooks.org

```
3   byte tmp;
4   byte i = 0;
5   byte j = 0;
6   bit old_gp_idle;
7   bit outermost;
8
9   do
10  :: i >= MAX_DYNTICK_LOOP_IRQ &&
11     j >= MAX_DYNTICK_LOOP_IRQ -> break;
12  :: i < MAX_DYNTICK_LOOP_IRQ ->
13    atomic {
14      outermost = (in_dyntick_irq == 0);
15      in_dyntick_irq = 1;
16    }
17 stmt1: skip;
18    atomic {
19      if
20      :: in_dyntick_nmi -> goto stmt1;
21      :: !in_dyntick_nmi && rcu_update_flag ->
22        goto stmt1_then;
23      :: else -> goto stmt1_else;
24      fi;
25    }
26 stmt1_then: skip;
27    EXECUTE_IRQ(stmt1_1, tmp = rcu_update_flag)
28    EXECUTE_IRQ(stmt1_2, rcu_update_flag = tmp + 1)
29 stmt1_else: skip;
30 stmt2: skip;  atomic {
31      if
32      :: in_dyntick_nmi -> goto stmt2;
33      :: !in_dyntick_nmi &&
34         !in_interrupt &&
35         (dynticks_progress_counter & 1) == 0 ->
36          goto stmt2_then;
37      :: else -> goto stmt2_else;
38      fi;
39    }
40 stmt2_then: skip;
41    EXECUTE_IRQ(stmt2_1,
42                tmp = dynticks_progress_counter)
43    EXECUTE_IRQ(stmt2_2,
44                dynticks_progress_counter = tmp + 1)
45    EXECUTE_IRQ(stmt2_3, tmp = rcu_update_flag)
46    EXECUTE_IRQ(stmt2_4, rcu_update_flag = tmp + 1)
47 stmt2_else: skip;
48    EXECUTE_IRQ(stmt3, tmp = in_interrupt)
49    EXECUTE_IRQ(stmt4, in_interrupt = tmp + 1)
50 stmt5: skip;
51    atomic {
52      if
53      :: in_dyntick_nmi -> goto stmt4;
54      :: !in_dyntick_nmi && outermost ->
55        old_gp_idle = (grace_period_state == GP_IDLE);
56      :: else -> skip;
57      fi;
58    }
59    i++;
60  :: j < i ->
61 stmt6: skip;
62    atomic {
63      if
64      :: in_dyntick_nmi -> goto stmt6;
65      :: !in_dyntick_nmi && j + 1 == i ->
66        assert(!old_gp_idle ||
67               grace_period_state != GP_DONE);
68      :: else -> skip;
69      fi;
70    }
71    EXECUTE_IRQ(stmt7, tmp = in_interrupt);
72    EXECUTE_IRQ(stmt8, in_interrupt = tmp - 1);
73 stmt9: skip;
74    atomic {
75      if
76      :: in_dyntick_nmi -> goto stmt9;
77      :: !in_dyntick_nmi && rcu_update_flag != 0 ->
78        goto stmt9_then;
79      :: else -> goto stmt9_else;
80      fi;
81    }
82 stmt9_then: skip;
83    EXECUTE_IRQ(stmt9_1, tmp = rcu_update_flag)
84    EXECUTE_IRQ(stmt9_2, rcu_update_flag = tmp - 1)
85 stmt9_3: skip;
86    atomic {
87      if
88      :: in_dyntick_nmi -> goto stmt9_3;
89      :: !in_dyntick_nmi && rcu_update_flag == 0 ->
90        goto stmt9_3_then;
91      :: else -> goto stmt9_3_else;
92      fi;
93    }
94 stmt9_3_then: skip;
95    EXECUTE_IRQ(stmt9_3_1,
96                tmp = dynticks_progress_counter)
97    EXECUTE_IRQ(stmt9_3_2,
98                dynticks_progress_counter = tmp + 1)
99 stmt9_3_else:
100 stmt9_else: skip;
101    atomic {
102      j++;
103      in_dyntick_irq = (i != j);
104    }
105  od;
106  dyntick_irq_done = 1;
107 }
```

Note that we have open-coded the "if" statements (for example, lines 17–29). In addition, statements that process strictly local state (such as line 59) need not exclude `dyntick_nmi()`.

Finally, `grace_period()` requires only a few changes:

```
1  proctype grace_period()
2  {
3    byte curr;
4    byte snap;
5    bit shouldexit;
6
7    grace_period_state = GP_IDLE;
8    atomic {
9      printf("MDL_NOHZ = %d\n", MAX_DYNTICK_LOOP_NOHZ);
10     printf("MDL_IRQ = %d\n", MAX_DYNTICK_LOOP_IRQ);
11     printf("MDL_NMI = %d\n", MAX_DYNTICK_LOOP_NMI);
12     shouldexit = 0;
13     snap = dynticks_progress_counter;
14     grace_period_state = GP_WAITING;
15   }
16   do
17   :: 1 ->
18     atomic {
19       assert(!shouldexit);
20       shouldexit = dyntick_nohz_done &&
21              dyntick_irq_done &&
22              dyntick_nmi_done;
23       curr = dynticks_progress_counter;
24       if
25       :: (curr - snap) >= 2 || (curr & 1) == 0 ->
26         break;
27       :: else -> skip;
28       fi;
29     }
30   od;
31   grace_period_state = GP_DONE;
32   grace_period_state = GP_IDLE;
33   atomic {
```

```
34      shouldexit = 0;
35      snap = dynticks_progress_counter;
36      grace_period_state = GP_WAITING;
37    }
38    do
39    :: 1 ->
40      atomic {
41        assert(!shouldexit);
42        shouldexit = dyntick_nohz_done &&
43               dyntick_irq_done &&
44               dyntick_nmi_done;
45        curr = dynticks_progress_counter;
46        if
47        :: (curr != snap) || ((curr & 1) == 0) ->
48          break;
49        :: else -> skip;
50        fi;
51      }
52    od;
53    grace_period_state = GP_DONE;
54  }
```

We have added the `printf()` for the new `MAX_DYNTICK_LOOP_NMI` parameter on line 11 and added `dyntick_nmi_done` to the `shouldexit` assignments on lines 22 and 44.

The model (`dyntickRCU-irq-nmi-ssl.spin`) results in a correct verification with several hundred million states, passing without errors.

Quick Quiz 12.21: Does Paul *always* write his code in this painfully incremental manner? ■

12.1.6.8 Lessons (Re)Learned

This effort provided some lessons (re)learned:

1. **Promela and Spin can verify interrupt/NMI-handler interactions.**

2. **Documenting code can help locate bugs.** In this case, the documentation effort located a misplaced memory barrier in `rcu_enter_nohz()` and `rcu_exit_nohz()`, as shown by the following patch [McK08b].

```
 static inline void rcu_enter_nohz(void)
 {
+       mb();
        __get_cpu_var(dynticks_progress_counter)++;
-       mb();
 }

 static inline void rcu_exit_nohz(void)
 {
-       mb();
        __get_cpu_var(dynticks_progress_counter)++;
+       mb();
 }
```

3. **Validate your code early, often, and up to the point of destruction.** This effort located one subtle bug in `rcu_try_flip_waitack_needed()` that would have been quite difficult to test or debug, as shown by the following patch [McK08c].

```
-       if ((curr - snap) > 2 || (snap & 0x1) == 0)
+       if ((curr - snap) > 2 || (curr & 0x1) == 0)
```

4. **Always verify your verification code.** The usual way to do this is to insert a deliberate bug and verify that the verification code catches it. Of course, if the verification code fails to catch this bug, you may also need to verify the bug itself, and so on, recursing infinitely. However, if you find yourself in this position, getting a good night's sleep can be an extremely effective debugging technique. You will then see that the obvious verify-the-verification technique is to deliberately insert bugs in the code being verified. If the verification fails to find them, the verification clearly is buggy.

5. **Use of atomic instructions can simplify verification.** Unfortunately, use of the `cmpxchg` atomic instruction would also slow down the critical IRQ fastpath, so they are not appropriate in this case.

6. **The need for complex formal verification often indicates a need to re-think your design.**

To this last point, it turns out that there is a much simpler solution to the dynticks problem, which is presented in the next section.

12.1.6.9 Simplicity Avoids Formal Verification

The complexity of the dynticks interface for preemptible RCU is primarily due to the fact that both IRQs and NMIs use the same code path and the same state variables. This leads to the notion of providing separate code paths and variables for IRQs and NMIs, as has been done for hierarchical RCU [McK08a] as indirectly suggested by Manfred Spraul [Spr08]. This work was pulled into mainline kernel during the v2.6.29 development cycle [McK08e].

12.1.6.10 State Variables for Simplified Dynticks Interface

Listing 12.17 shows the new per-CPU state variables. These variables are grouped into structs to allow multiple independent RCU implementations (e.g., `rcu` and `rcu_bh`) to conveniently and efficiently share dynticks state.

www.dbooks.org

Listing 12.17: Variables for Simple Dynticks Interface

```
1 struct rcu_dynticks {
2   int dynticks_nesting;
3   int dynticks;
4   int dynticks_nmi;
5 };
6
7 struct rcu_data {
8   ...
9   int dynticks_snap;
10  int dynticks_nmi_snap;
11  ...
12 };
```

In what follows, they can be thought of as independent per-CPU variables.

The `dynticks_nesting`, `dynticks`, and `dynticks_snap` variables are for the IRQ code paths, and the `dynticks_nmi` and `dynticks_nmi_snap` variables are for the NMI code paths, although the NMI code path will also reference (but not modify) the `dynticks_nesting` variable. These variables are used as follows:

`dynticks_nesting`
: This counts the number of reasons that the corresponding CPU should be monitored for RCU read-side critical sections. If the CPU is in dynticks-idle mode, then this counts the IRQ nesting level, otherwise it is one greater than the IRQ nesting level.

`dynticks`
: This counter's value is even if the corresponding CPU is in dynticks-idle mode and there are no IRQ handlers currently running on that CPU, otherwise the counter's value is odd. In other words, if this counter's value is odd, then the corresponding CPU might be in an RCU read-side critical section.

`dynticks_nmi`
: This counter's value is odd if the corresponding CPU is in an NMI handler, but only if the NMI arrived while this CPU was in dyntick-idle mode with no IRQ handlers running. Otherwise, the counter's value will be even.

`dynticks_snap`
: This will be a snapshot of the `dynticks` counter, but only if the current RCU grace period has extended for too long a duration.

`dynticks_nmi_snap`
: This will be a snapshot of the `dynticks_nmi` counter, but again only if the current RCU grace period has extended for too long a duration.

Listing 12.18: Entering and Exiting Dynticks-Idle Mode

```
1 void rcu_enter_nohz(void)
2 {
3   unsigned long flags;
4   struct rcu_dynticks *rdtp;
5
6   smp_mb();
7   local_irq_save(flags);
8   rdtp = &__get_cpu_var(rcu_dynticks);
9   rdtp->dynticks++;
10  rdtp->dynticks_nesting--;
11  WARN_ON(rdtp->dynticks & 0x1);
12  local_irq_restore(flags);
13 }
14
15 void rcu_exit_nohz(void)
16 {
17  unsigned long flags;
18  struct rcu_dynticks *rdtp;
19
20  local_irq_save(flags);
21  rdtp = &__get_cpu_var(rcu_dynticks);
22  rdtp->dynticks++;
23  rdtp->dynticks_nesting++;
24  WARN_ON(!(rdtp->dynticks & 0x1));
25  local_irq_restore(flags);
26  smp_mb();
27 }
```

If both `dynticks` and `dynticks_nmi` have taken on an even value during a given time interval, then the corresponding CPU has passed through a quiescent state during that interval.

Quick Quiz 12.22: But what happens if an NMI handler starts running before an IRQ handler completes, and if that NMI handler continues running until a second IRQ handler starts? ■

12.1.6.11 Entering and Leaving Dynticks-Idle Mode

Listing 12.18 shows the `rcu_enter_nohz()` and `rcu_exit_nohz()`, which enter and exit dynticks-idle mode, also known as "nohz" mode. These two functions are invoked from process context.

Line 6 ensures that any prior memory accesses (which might include accesses from RCU read-side critical sections) are seen by other CPUs before those marking entry to dynticks-idle mode. Lines 7 and 12 disable and reenable IRQs. Line 8 acquires a pointer to the current CPU's `rcu_dynticks` structure, and line 9 increments the current CPU's `dynticks` counter, which should now be even, given that we are entering dynticks-idle mode in process context. Finally, line 10 decrements `dynticks_nesting`, which should now be zero.

The `rcu_exit_nohz()` function is quite similar, but increments `dynticks_nesting` rather than decrementing it and checks for the opposite `dynticks` polarity.

Listing 12.19: NMIs From Dynticks-Idle Mode

```
1  void rcu_nmi_enter(void)
2  {
3    struct rcu_dynticks *rdtp;
4
5    rdtp = &__get_cpu_var(rcu_dynticks);
6    if (rdtp->dynticks & 0x1)
7      return;
8    rdtp->dynticks_nmi++;
9    WARN_ON(!(rdtp->dynticks_nmi & 0x1));
10   smp_mb();
11 }
12
13 void rcu_nmi_exit(void)
14 {
15   struct rcu_dynticks *rdtp;
16
17   rdtp = &__get_cpu_var(rcu_dynticks);
18   if (rdtp->dynticks & 0x1)
19     return;
20   smp_mb();
21   rdtp->dynticks_nmi++;
22   WARN_ON(rdtp->dynticks_nmi & 0x1);
23 }
```

Listing 12.20: Interrupts From Dynticks-Idle Mode

```
1  void rcu_irq_enter(void)
2  {
3    struct rcu_dynticks *rdtp;
4
5    rdtp = &__get_cpu_var(rcu_dynticks);
6    if (rdtp->dynticks_nesting++)
7      return;
8    rdtp->dynticks++;
9    WARN_ON(!(rdtp->dynticks & 0x1));
10   smp_mb();
11 }
12
13 void rcu_irq_exit(void)
14 {
15   struct rcu_dynticks *rdtp;
16
17   rdtp = &__get_cpu_var(rcu_dynticks);
18   if (--rdtp->dynticks_nesting)
19     return;
20   smp_mb();
21   rdtp->dynticks++;
22   WARN_ON(rdtp->dynticks & 0x1);
23   if (__get_cpu_var(rcu_data).nxtlist ||
24       __get_cpu_var(rcu_bh_data).nxtlist)
25     set_need_resched();
26 }
```

12.1.6.12 NMIs From Dynticks-Idle Mode

Listing 12.19 shows the `rcu_nmi_enter()` and `rcu_nmi_exit()` functions, which inform RCU of NMI entry and exit, respectively, from dynticks-idle mode. However, if the NMI arrives during an IRQ handler, then RCU will already be on the lookout for RCU read-side critical sections from this CPU, so lines 6 and 7 of `rcu_nmi_enter()` and lines 18 and 19 of `rcu_nmi_exit()` silently return if `dynticks` is odd. Otherwise, the two functions increment `dynticks_nmi`, with `rcu_nmi_enter()` leaving it with an odd value and `rcu_nmi_exit()` leaving it with an even value. Both functions execute memory barriers between this increment and possible RCU read-side critical sections on lines 10 and 20, respectively.

12.1.6.13 Interrupts From Dynticks-Idle Mode

Listing 12.20 shows `rcu_irq_enter()` and `rcu_irq_exit()`, which inform RCU of entry to and exit from, respectively, IRQ context. Line 6 of `rcu_irq_enter()` increments `dynticks_nesting`, and if this variable was already non-zero, line 7 silently returns. Otherwise, line 8 increments `dynticks`, which will then have an odd value, consistent with the fact that this CPU can now execute RCU read-side critical sections. Line 10 therefore executes a memory barrier to ensure that the increment of `dynticks` is seen before any RCU read-side critical sections that the subsequent IRQ handler might execute.

Line 18 of `rcu_irq_exit()` decrements `dynticks_nesting`, and if the result is non-zero, line 19 silently returns. Otherwise, line 20 executes a memory barrier to ensure that the increment of `dynticks` on line 21 is seen after any RCU read-side critical sections that the prior IRQ handler might have executed. Line 22 verifies that `dynticks` is now even, consistent with the fact that no RCU read-side critical sections may appear in dynticks-idle mode. Lines 23–25 check to see if the prior IRQ handlers enqueued any RCU callbacks, forcing this CPU out of dynticks-idle mode via a reschedule API if so.

12.1.6.14 Checking For Dynticks Quiescent States

Listing 12.21 shows `dyntick_save_progress_counter()`, which takes a snapshot of the specified CPU's `dynticks` and `dynticks_nmi` counters. Lines 8 and 9 snapshot these two variables to locals, line 10 executes a memory barrier to pair with the memory barriers in the functions in Listings 12.18, 12.19, and 12.20. Lines 11 and 12 record the snapshots for later calls to `rcu_implicit_dynticks_qs()`, and line 13 checks to see if the CPU is in dynticks-idle mode with neither IRQs nor NMIs in progress (in other words, both snapshots have even values), hence in an extended quiescent state. If so, lines 14 and 15 count this event, and line 16 returns true if the CPU was in a quiescent state.

Listing 12.22 shows `rcu_implicit_dynticks_qs()`, which is called to check whether a CPU has entered dyntick-idle mode subsequent to a call to `dynticks_save_progress_counter()`. Lines 9 and 11 take new snapshots of the corresponding CPU's `dynticks` and

www.dbooks.org

Listing 12.21: Saving Dyntick Progress Counters

```
1 static int
2 dyntick_save_progress_counter(struct rcu_data *rdp)
3 {
4   int ret;
5   int snap;
6   int snap_nmi;
7
8   snap = rdp->dynticks->dynticks;
9   snap_nmi = rdp->dynticks->dynticks_nmi;
10  smp_mb();
11  rdp->dynticks_snap = snap;
12  rdp->dynticks_nmi_snap = snap_nmi;
13  ret = ((snap & 0x1) == 0) && ((snap_nmi & 0x1) == 0);
14  if (ret)
15    rdp->dynticks_fqs++;
16  return ret;
17 }
```

Listing 12.22: Checking Dyntick Progress Counters

```
1 static int
2 rcu_implicit_dynticks_qs(struct rcu_data *rdp)
3 {
4   long curr;
5   long curr_nmi;
6   long snap;
7   long snap_nmi;
8
9   curr = rdp->dynticks->dynticks;
10  snap = rdp->dynticks_snap;
11  curr_nmi = rdp->dynticks->dynticks_nmi;
12  snap_nmi = rdp->dynticks_nmi_snap;
13  smp_mb();
14  if ((curr != snap || (curr & 0x1) == 0) &&
15      (curr_nmi != snap_nmi || (curr_nmi & 0x1) == 0)) {
16    rdp->dynticks_fqs++;
17    return 1;
18  }
19  return rcu_implicit_offline_qs(rdp);
20 }
```

`dynticks_nmi` variables, while lines 10 and 12 retrieve the snapshots saved earlier by `dynticks_save_progress_counter()`. Line 13 then executes a memory barrier to pair with the memory barriers in the functions in Listings 12.18, 12.19, and 12.20. Lines 14–15 then check to see if the CPU is either currently in a quiescent state (`curr` and `curr_nmi` having even values) or has passed through a quiescent state since the last call to `dynticks_save_progress_counter()` (the values of `dynticks` and `dynticks_nmi` having changed). If these checks confirm that the CPU has passed through a dyntick-idle quiescent state, then line 16 counts that fact and line 17 returns an indication of this fact. Either way, line 19 checks for race conditions that can result in RCU waiting for a CPU that is offline.

Quick Quiz 12.23: This is still pretty complicated. Why not just have a `cpumask_t` with per-CPU bits, clearing the bit when entering an IRQ or NMI handler, and setting it upon exit? ■

Linux-kernel RCU's dyntick-idle code has since been rewritten yet again based on a suggestion from Andy Lutomirski [McK15b], but it is time to sum up and move on to other topics.

12.1.6.15 Discussion

A slight shift in viewpoint resulted in a substantial simplification of the dynticks interface for RCU. The key change leading to this simplification was minimizing of sharing between IRQ and NMI contexts. The only sharing in this simplified interface is references from NMI context to IRQ variables (the `dynticks` variable). This type of sharing is benign, because the NMI functions never update this variable, so that its value remains constant through the lifetime of the NMI handler. This limitation of sharing allows the individual functions to be understood one at a time, in happy contrast to the situation described in Section 12.1.5, where an NMI might change shared state at any point during execution of the IRQ functions.

Verification can be a good thing, but simplicity is even better.

12.2 Special-Purpose State-Space Search

> Jack of all trades, master of none.
>
> *Unknown*

Although Promela and spin allow you to verify pretty much any (smallish) algorithm, their very generality can sometimes be a curse. For example, Promela does not understand memory models or any sort of reordering semantics. This section therefore describes some state-space search tools that understand memory models used by production systems, greatly simplifying the verification of weakly ordered code.

For example, Section 12.1.4 showed how to convince Promela to account for weak memory ordering. Although this approach can work well, it requires that the developer fully understand the system's memory model. Unfortunately, few (if any) developers fully understand the complex memory models of modern CPUs.

Therefore, another approach is to use a tool that already understands this memory ordering, such as the PPCMEM tool produced by Peter Sewell and Susmit Sarkar at the University of Cambridge, Luc Maranget, Francesco Zappa Nardelli, and Pankaj Pawan at INRIA, and Jade Alglave

Listing 12.23: PPCMEM Litmus Test

```
1  PPC SB+lwsync-RMW-lwsync+isync-simple
2  ""
3  {
4  0:r2=x; 0:r3=2; 0:r4=y; 0:r10=0; 0:r11=0; 0:r12=z;
5  1:r2=y; 1:r4=x;
6  }
7   P0                  | P1           ;
8   li r1,1             | li r1,1      ;
9   stw r1,0(r2)        | stw r1,0(r2) ;
10  lwsync              | sync         ;
11                      | lwz r3,0(r4) ;
12  lwarx  r11,r10,r12 | ;
13  stwcx. r11,r10,r12 | ;
14  bne Fail1           | ;
15  isync               | ;
16  lwz r3,0(r4)        | ;
17  Fail1:              | ;
18
19 exists
20 (0:r3=0 /\ 1:r3=0)
```

at Oxford University, in cooperation with Derek Williams of IBM [AMP+11]. This group formalized the memory models of Power, Arm, x86, as well as that of the C/C++11 standard [Smi19], and produced the PPCMEM tool based on the Power and Arm formalizations.

Quick Quiz 12.24: But x86 has strong memory ordering, so why formalize its memory model? ■

The PPCMEM tool takes *litmus tests* as input. A sample litmus test is presented in Section 12.2.1. Section 12.2.2 relates this litmus test to the equivalent C-language program, Section 12.2.3 describes how to apply PPCMEM to this litmus test, and Section 12.2.4 discusses the implications.

12.2.1 Anatomy of a Litmus Test

An example PowerPC litmus test for PPCMEM is shown in Listing 12.23. The ARM interface works the same way, but with Arm instructions substituted for the Power instructions and with the initial "PPC" replaced by "ARM".

In the example, line 1 identifies the type of system ("ARM" or "PPC") and contains the title for the model. Line 2 provides a place for an alternative name for the test, which you will usually want to leave blank as shown in the above example. Comments can be inserted between lines 2 and 3 using the OCaml (or Pascal) syntax of `(* *)`.

Lines 3–6 give initial values for all registers; each is of the form `P:R=V`, where `P` is the process identifier, `R` is the register identifier, and `V` is the value. For example, process 0's register r3 initially contains the value 2. If the value is a variable (`x`, `y`, or `z` in the example) then the register is initialized to the address of the variable. It is also possible to initialize the contents of variables, for example, `x=1` initializes the value of `x` to 1. Uninitialized variables default to the value zero, so that in the example, `x`, `y`, and `z` are all initially zero.

Line 7 provides identifiers for the two processes, so that the `0:r3=2` on line 4 could instead have been written `P0:r3=2`. Line 7 is required, and the identifiers must be of the form `Pn`, where `n` is the column number, starting from zero for the left-most column. This may seem unnecessarily strict, but it does prevent considerable confusion in actual use.

Quick Quiz 12.25: Why does line 8 of Listing 12.23 initialize the registers? Why not instead initialize them on lines 4 and 5? ■

Lines 8–17 are the lines of code for each process. A given process can have empty lines, as is the case for P0's line 11 and P1's lines 12–17. Labels and branches are permitted, as demonstrated by the branch on line 14 to the label on line 17. That said, too-free use of branches will expand the state space. Use of loops is a particularly good way to explode your state space.

Lines 19–20 show the assertion, which in this case indicates that we are interested in whether P0's and P1's r3 registers can both contain zero after both threads complete execution. This assertion is important because there are a number of use cases that would fail miserably if both P0 and P1 saw zero in their respective r3 registers.

This should give you enough information to construct simple litmus tests. Some additional documentation is available, though much of this additional documentation is intended for a different research tool that runs tests on actual hardware. Perhaps more importantly, a large number of pre-existing litmus tests are available with the online tool (available via the "Select ARM Test" and "Select POWER Test" buttons at `https://www.cl.cam.ac.uk/~pes20/ppcmem/`). It is quite likely that one of these pre-existing litmus tests will answer your Power or Arm memory-ordering question.

12.2.2 What Does This Litmus Test Mean?

P0's lines 8 and 9 are equivalent to the C statement `x=1` because line 4 defines P0's register `r2` to be the address of `x`. P0's lines 12 and 13 are the mnemonics for load-linked ("load register exclusive" in Arm parlance and "load reserve" in Power parlance) and store-conditional ("store register exclusive" in Arm parlance), respectively. When these are used together, they form an atomic instruction sequence, roughly similar to the compare-and-swap sequences exemplified by the x86 `lock;cmpxchg`

www.dbooks.org

Listing 12.24: Meaning of PPCMEM Litmus Test

```
1 void P0(void)
2 {
3   int r3;
4
5   x = 1; /* Lines 8 and 9 */
6   atomic_add_return(&z, 0); /* Lines 10-15 */
7   r3 = y; /* Line 16 */
8 }
9
10 void P1(void)
11 {
12   int r3;
13
14   y = 1; /* Lines 8-9 */
15   smp_mb(); /* Line 10 */
16   r3 = x; /* Line 11 */
17 }
```

instruction. Moving to a higher level of abstraction, the sequence from lines 10–15 is equivalent to the Linux kernel's `atomic_add_return(&z, 0)`. Finally, line 16 is roughly equivalent to the C statement `r3=y`.

P1's lines 8 and 9 are equivalent to the C statement `y=1`, line 10 is a memory barrier, equivalent to the Linux kernel statement `smp_mb()`, and line 11 is equivalent to the C statement `r3=x`.

Quick Quiz 12.26: But whatever happened to line 17 of Listing 12.23, the one that is the `Fail1:` label? ■

Putting all this together, the C-language equivalent to the entire litmus test is as shown in Listing 12.24. The key point is that if `atomic_add_return()` acts as a full memory barrier (as the Linux kernel requires it to), then it should be impossible for `P0()`'s and `P1()`'s `r3` variables to both be zero after execution completes.

The next section describes how to run this litmus test.

12.2.3 Running a Litmus Test

As noted earlier, litmus tests may be run interactively via `https://www.cl.cam.ac.uk/~pes20/ppcmem/`, which can help build an understanding of the memory model. However, this approach requires that the user manually carry out the full state-space search. Because it is very difficult to be sure that you have checked every possible sequence of events, a separate tool is provided for this purpose [McK11d].

Because the litmus test shown in Listing 12.23 contains read-modify-write instructions, we must add `-model` arguments to the command line. If the litmus test is stored in `filename.litmus`, this will result in the output shown in Listing 12.25, where the `...` stands for voluminous making-progress output. The list of states in-

Listing 12.25: PPCMEM Detects an Error

```
./ppcmem -model lwsync_read_block \
         -model coherence_points filename.litmus
...
States 6
0:r3=0; 1:r3=0;
0:r3=0; 1:r3=1;
0:r3=1; 1:r3=0;
0:r3=1; 1:r3=1;
0:r3=2; 1:r3=0;
0:r3=2; 1:r3=1;
Ok
Condition exists (0:r3=0 /\ 1:r3=0)
Hash=e2240ce2072a2610c034ccd4fc964e77
Observation SB+lwsync-RMW-lwsync+isync Sometimes 1
```

Listing 12.26: PPCMEM on Repaired Litmus Test

```
./ppcmem -model lwsync_read_block \
         -model coherence_points filename.litmus
...
States 5
0:r3=0; 1:r3=1;
0:r3=1; 1:r3=0;
0:r3=1; 1:r3=1;
0:r3=2; 1:r3=0;
0:r3=2; 1:r3=1;
No (allowed not found)
Condition exists (0:r3=0 /\ 1:r3=0)
Hash=77dd723cda9981248ea4459fcdf6097d
Observation SB+lwsync-RMW-lwsync+sync Never 0 5
```

cludes `0:r3=0; 1:r3=0;`, indicating once again that the old PowerPC implementation of `atomic_add_return()` does not act as a full barrier. The "Sometimes" on the last line confirms this: the assertion triggers for some executions, but not all of the time.

The fix to this Linux-kernel bug is to replace P0's `isync` with `sync`, which results in the output shown in Listing 12.26. As you can see, `0:r3=0; 1:r3=0;` does not appear in the list of states, and the last line calls out "Never". Therefore, the model predicts that the offending execution sequence cannot happen.

Quick Quiz 12.27: Does the Arm Linux kernel have a similar bug? ■

Quick Quiz 12.28: Does the `lwsync` on line 10 in Listing 12.23 provide sufficient ordering? ■

12.2.4 PPCMEM Discussion

These tools promise to be of great help to people working on low-level parallel primitives that run on Arm and on Power. These tools do have some intrinsic limitations:

1. These tools are research prototypes, and as such are unsupported.

2. These tools do not constitute official statements by IBM or Arm on their respective CPU architectures. For example, both corporations reserve the right to report a bug at any time against any version of any of these tools. These tools are therefore not a substitute for careful stress testing on real hardware. Moreover, both the tools and the model that they are based on are under active development and might change at any time. On the other hand, this model was developed in consultation with the relevant hardware experts, so there is good reason to be confident that it is a robust representation of the architectures.

3. These tools currently handle a subset of the instruction set. This subset has been sufficient for my purposes, but your mileage may vary. In particular, the tool handles only word-sized accesses (32 bits), and the words accessed must be properly aligned.[3] In addition, the tool does not handle some of the weaker variants of the Arm memory-barrier instructions, nor does it handle arithmetic.

4. The tools are restricted to small loop-free code fragments running on small numbers of threads. Larger examples result in state-space explosion, just as with similar tools such as Promela and spin.

5. The full state-space search does not give any indication of how each offending state was reached. That said, once you realize that the state is in fact reachable, it is usually not too hard to find that state using the interactive tool.

6. These tools are not much good for complex data structures, although it is possible to create and traverse extremely simple linked lists using initialization statements of the form "`x=y; y=z; z=42;`".

7. These tools do not handle memory mapped I/O or device registers. Of course, handling such things would require that they be formalized, which does not appear to be in the offing.

8. The tools will detect only those problems for which you code an assertion. This weakness is common to all formal methods, and is yet another reason why testing remains important. In the immortal words of Donald Knuth quoted at the beginning of this chapter, "Beware of bugs in the above code; I have only proved it correct, not tried it."

[3] But recent work focuses on mixed-size accesses [FSP+17].

That said, one strength of these tools is that they are designed to model the full range of behaviors allowed by the architectures, including behaviors that are legal, but which current hardware implementations do not yet inflict on unwary software developers. Therefore, an algorithm that is vetted by these tools likely has some additional safety margin when running on real hardware. Furthermore, testing on real hardware can only find bugs; such testing is inherently incapable of proving a given usage correct. To appreciate this, consider that the researchers routinely ran in excess of 100 billion test runs on real hardware to validate their model. In one case, behavior that is allowed by the architecture did not occur, despite 176 billion runs [AMP+11]. In contrast, the full-state-space search allows the tool to prove code fragments correct.

It is worth repeating that formal methods and tools are no substitute for testing. The fact is that producing large reliable concurrent software artifacts, the Linux kernel for example, is quite difficult. Developers must therefore be prepared to apply every tool at their disposal towards this goal. The tools presented in this chapter are able to locate bugs that are quite difficult to produce (let alone track down) via testing. On the other hand, testing can be applied to far larger bodies of software than the tools presented in this chapter are ever likely to handle. As always, use the right tools for the job!

Of course, it is always best to avoid the need to work at this level by designing your parallel code to be easily partitioned and then using higher-level primitives (such as locks, sequence counters, atomic operations, and RCU) to get your job done more straightforwardly. And even if you absolutely must use low-level memory barriers and read-modify-write instructions to get your job done, the more conservative your use of these sharp instruments, the easier your life is likely to be.

12.3 Axiomatic Approaches

> Theory helps us to bear our ignorance of facts.
>
> *George Santayana*

Although the PPCMEM tool can solve the famous "independent reads of independent writes" (IRIW) litmus test shown in Listing 12.27, doing so requires no less than fourteen CPU hours and generates no less than ten gigabytes of state space. That said, this situation is a great improvement over that before the advent of PPCMEM, where solving this problem required perusing volumes

www.dbooks.org

Listing 12.27: IRIW Litmus Test

```
1  PPC IRIW.litmus
2  ""
3  (* Traditional IRIW. *)
4  {
5  0:r1=1; 0:r2=x;
6  1:r1=1;         1:r4=y;
7          2:r2=x; 2:r4=y;
8          3:r2=x; 3:r4=y;
9  }
10 P0           | P1           | P2           | P3           ;
11 stw r1,0(r2) | stw r1,0(r4) | lwz r3,0(r2) | lwz r3,0(r4) ;
12              |              | sync         | sync         ;
13              |              | lwz r5,0(r4) | lwz r5,0(r2) ;
14
15 exists
16 (2:r3=1 /\ 2:r5=0 /\ 3:r3=1 /\ 3:r5=0)
```

Listing 12.28: Expanded IRIW Litmus Test

```
1  PPC IRIW5.litmus
2  ""
3  (* Traditional IRIW, but with five stores instead of *)
4  (* just one.                                          *)
5  {
6  0:r1=1; 0:r2=x;
7  1:r1=1;         1:r4=y;
8          2:r2=x; 2:r4=y;
9          3:r2=x; 3:r4=y;
10 }
11 P0           | P1           | P2           | P3           ;
12 stw r1,0(r2) | stw r1,0(r4) | lwz r3,0(r2) | lwz r3,0(r4) ;
13 addi r1,r1,1 | addi r1,r1,1 | sync         | sync         ;
14 stw r1,0(r2) | stw r1,0(r4) | lwz r5,0(r4) | lwz r5,0(r2) ;
15 addi r1,r1,1 | addi r1,r1,1 |              |              ;
16 stw r1,0(r2) | stw r1,0(r4) |              |              ;
17 addi r1,r1,1 | addi r1,r1,1 |              |              ;
18 stw r1,0(r2) | stw r1,0(r4) |              |              ;
19 addi r1,r1,1 | addi r1,r1,1 |              |              ;
20 stw r1,0(r2) | stw r1,0(r4) |              |              ;
21
22 exists
23 (2:r3=1 /\ 2:r5=0 /\ 3:r3=1 /\ 3:r5=0)
```

of reference manuals, attempting proofs, discussing with experts, and being unsure of the final answer. Although fourteen hours can seem like a long time, it is much shorter than weeks or even months.

However, the time required is a bit surprising given the simplicity of the litmus test, which has two threads storing to two separate variables and two other threads loading from these two variables in opposite orders. The assertion triggers if the two loading threads disagree on the order of the two stores. Even by the standards of memory-order litmus tests, this is quite simple.

One reason for the amount of time and space consumed is that PPCMEM does a trace-based full-state-space search, which means that it must generate and evaluate all possible orders and combinations of events at the architectural level. At this level, both loads and stores correspond to ornate sequences of events and actions, resulting in a very large state space that must be completely searched, in turn resulting in large memory and CPU consumption.

Of course, many of the traces are quite similar to one another, which suggests that an approach that treated similar traces as one might improve performace. One such approach is the axiomatic approach of Alglave et al. [AMT14], which creates a set of axioms to represent the memory model and then converts litmus tests to theorems that might be proven or disproven over this set of axioms. The resulting tool, called "herd", conveniently takes as input the same litmus tests as PPCMEM, including the IRIW litmus test shown in Listing 12.27.

However, where PPCMEM requires 14 CPU hours to solve IRIW, herd does so in 17 milliseconds, which represents a speedup of more than six orders of magnitude. That said, the problem is exponential in nature, so we should expect herd to exhibit exponential slowdowns for larger problems. And this is exactly what happens, for example, if we add four more writes per writing CPU as shown in Listing 12.28, herd slows down by a factor of more than 50,000, requiring more than 15 *minutes* of CPU time. Adding threads also results in exponential slowdowns [MS14].

Despite their exponential nature, both PPCMEM and herd have proven quite useful for checking key parallel algorithms, including the queued-lock handoff on x86 systems. The weaknesses of the herd tool are similar to those of PPCMEM, which were described in Section 12.2.4. There are some obscure (but very real) cases for which the PPCMEM and herd tools disagree, and as of late 2014 resolving these disagreements was ongoing.

It would be helpful if the litmus tests could be written in C (as in Listing 12.24) rather than assembly (as in Listing 12.23). This is now possible, as will be described in the following sections.

12.3.1 Axiomatic Approaches and Locking

Axiomatic approaches may also be applied to higher-level languages and also to higher-level synchronization primitives, as exemplified by the lock-based litmus test shown in Listing 12.29 (`C-Lock1.litmus`). This litmus test can be modeled by the Linux kernel memory model (LKMM) [AMM+18, MS18]. As expected, the `herd` tool's output features the string `Never`, correctly indicating that `P1()` cannot see `x` having a value of one.[4]

[4] The output of the `herd` tool is compatible with that of PPCMEM, so feel free to look at Listings 12.25 and 12.26 for examples showing the output format.

Listing 12.29: Locking Example

```
1 C Lock1
2
3 {}
4
5 P0(int *x, spinlock_t *sp)
6 {
7   spin_lock(sp);
8   WRITE_ONCE(*x, 1);
9   WRITE_ONCE(*x, 0);
10  spin_unlock(sp);
11 }
12
13 P1(int *x, spinlock_t *sp)
14 {
15  int r1;
16
17  spin_lock(sp);
18  r1 = READ_ONCE(*x);
19  spin_unlock(sp);
20 }
21
22 exists (1:r1=1)
```

Listing 12.30: Broken Locking Example

```
1 C Lock2
2
3 {}
4
5 P0(int *x, spinlock_t *sp1)
6 {
7   spin_lock(sp1);
8   WRITE_ONCE(*x, 1);
9   WRITE_ONCE(*x, 0);
10  spin_unlock(sp1);
11 }
12
13 P1(int *x, spinlock_t *sp2) // Buggy!
14 {
15  int r1;
16
17  spin_lock(sp2);
18  r1 = READ_ONCE(*x);
19  spin_unlock(sp2);
20 }
21
22 exists (1:r1=1)
```

Quick Quiz 12.29: What do you have to do to run `herd` on litmus tests like that shown in Listing 12.29? ■

Of course, if `P0()` and `P1()` use different locks, as shown in Listing 12.30 (`C-Lock2.litmus`), then all bets are off. And in this case, the `herd` tool's output features the string `Sometimes`, correctly indicating that use of different locks allows `P1()` to see `x` having a value of one.

Quick Quiz 12.30: Why bother modeling locking directly? Why not simply emulate locking with atomic operations? ■

But locking is not the only synchronization primitive that can be modeled directly: The next section looks at RCU.

Listing 12.31: Canonical RCU Removal Litmus Test

```
1 C C-RCU-remove
2
3 {
4   int z=1;
5   int y=2;
6   int *x=y;
7 }
8
9 P0(int **x, int *y, int *z)
10 {
11  rcu_assign_pointer(*x, z);
12  synchronize_rcu();
13  WRITE_ONCE(*y, 0);
14 }
15
16 P1(int **x, int *y, int *z)
17 {
18  int *r1;
19  int r2;
20
21  rcu_read_lock();
22  r1 = rcu_dereference(*x);
23  r2 = READ_ONCE(*r1);
24  rcu_read_unlock();
25 }
26
27 locations [1:r1; x; y; z]
28 exists (1:r2=0)
```

12.3.2 Axiomatic Approaches and RCU

Axiomatic approaches can also analyze litmus tests involving RCU [AMM+18]. To that end, Listing 12.31 (`C-RCU-remove.litmus`) shows a litmus test corresponding to the canonical RCU-mediated removal from a linked list. As with the locking litmus test, this RCU litmus test can be modeled by LKMM, with similar performance advantages compared to modeling emulations of RCU. Line 6 shows `x` as the list head, initially referencing `y`, which in turn is initialized to the value 2 on line 5.

`P0()` on lines 9–14 removes element `y` from the list by replacing it with element `z` (line 11), waits for a grace period (line 12), and finally zeroes `y` to emulate `free()` (line 13). `P1()` on lines 16–25 executes within an RCU read-side critical section (lines 21–24), picking up the list head (line 22) and then loading the next element (line 23). The next element should be non-zero, that is, not yet freed (line 28). Several other variables are output for debugging purposes (line 27).

The output of the `herd` tool when running this litmus test features `Never`, indicating that `P0()` never accesses a freed element, as expected. Also as expected, removing line 12 results in `P0()` accessing a freed element, as indicated by the `Sometimes` in the `herd` output.

A litmus test for a more complex example proposed by Roman Penyaev [Pen18] is shown in Listing 12.32 (`C-RomanPenyaev-list-rcu-rr.litmus`). In this ex-

www.dbooks.org

Listing 12.32: Complex RCU Litmus Test

```
1  C C-RomanPenyaev-list-rcu-rr
2
3  {
4    int *z=1;
5    int *y=z;
6    int *x=y;
7    int *w=x;
8    int *v=w;
9    int *c=w;
10 }
11
12 P0(int **c, int **v)
13 {
14   int *r1;
15   int *r2;
16   int *r3;
17   int *r4;
18
19   rcu_read_lock();
20   r1 = READ_ONCE(*c);
21   if (r1 == 0) {
22     r1 = READ_ONCE(*v);
23   }
24   r2 = rcu_dereference(*(int **)r1);
25   smp_store_release(c, r2);
26   rcu_read_unlock();
27   rcu_read_lock();
28   r3 = READ_ONCE(*c);
29   if (r3 == 0) {
30     r3 = READ_ONCE(*v);
31   }
32   r4 = rcu_dereference(*(int **)r3);
33   smp_store_release(c, r4);
34   rcu_read_unlock();
35 }
36
37 P1(int **c, int **v, int **w, int **x, int **y)
38 {
39   int *r1;
40
41   rcu_assign_pointer(*w, y);
42   synchronize_rcu();
43   r1 = READ_ONCE(*c);
44   if ((int **)r1 == x) {
45     WRITE_ONCE(*c, 0);
46     synchronize_rcu();
47   }
48   smp_store_release(x, 0);
49 }
50
51 locations [1:r1; c; v; w; x; y]
52 exists (0:r1=0 \/ 0:r2=0 \/ 0:r3=0 \/ 0:r4=0)
```

ample, readers (modeled by `P0()` on lines 12–35) access a linked list in a round-robin fashion by "leaking" a pointer to the last list element accessed into variable `c`. Updaters (modeled by `P1()` on lines 37–49) remove an element, taking care to avoid disrupting current or future readers.

Quick Quiz 12.31: Wait!!! Isn't leaking pointers out of an RCU read-side critical section a critical bug??? ■

Lines 4–8 define the initial linked list, tail first. In the Linux kernel, this would be a doubly linked circular list, but `herd` is currently incapable of modeling such a beast. The strategy is instead to use a singly linked linear list that is long enough that the end is never reached. Line 9 defines variable `c`, which is used to cache the list pointer between successive RCU read-side critical sections.

Again, `P0()` on lines 12–35 models readers. This process models a pair of successive readers traversing round-robin through the list, with the first reader on lines 19–26 and the second reader on lines 27–34. Line 20 fetches the pointer cached in `c`, and if line 21 sees that the pointer was `NULL`, line 22 restarts at the beginning of the list. In either case, line 24 advances to the next list element, and line 25 stores a pointer to this element back into variable `c`. Lines 27–34 repeat this process, but using registers `r3` and `r4` instead of `r1` and `r2`. As with Listing 12.31, this litmus test stores zero to emulate `free()`, so line 52 checks for any of these four registers being `NULL`, also known as zero.

Because `P0()` leaks an RCU-protected pointer from its first RCU read-side critical section to its second, `P1()` must carry out its update (removing `x`) very carefully. Line 41 removes `x` by linking `w` to `y`. Line 42 waits for readers, after which no subsequent reader has a path to `x` via the linked list. Line 43 fetches `c`, and if line 44 determines that `c` references the newly removed `x`, line 45 sets `c` to `NULL` and line 46 again waits for readers, after which no subsequent reader can fetch `x` from `c`. In either case, line 48 emulates `free()` by storing zero to `x`.

Quick Quiz 12.32: In Listing 12.32, why couldn't a reader fetch `c` just before `P1()` zeroed it on line 45, and then later store this same value back into `c` just after it was zeroed, thus defeating the zeroing operation? ■

The output of the `herd` tool when running this litmus test features `Never`, indicating that `P0()` never accesses a freed element, as expected. Also as expected, removing either `synchronize_rcu()` results in `P1()` accessing a freed element, as indicated by `Sometimes` in the `herd` output.

Quick Quiz 12.33: In Listing 12.32, why not have just one call to `synchronize_rcu()` immediately before line 48? ■

Quick Quiz 12.34: Also in Listing 12.32, can't line 48 be `WRITE_ONCE()` instead of `smp_store_release()`? ■

These sections have shown how axiomatic approaches can successfully model synchronization primitives such as locking and RCU in C-language litmus tests. Longer term, the hope is that the axiomatic approaches will model even higher-level software artifacts, producing exponential verification speedups. This could potentially allow axiomatic verification of much larger software systems.

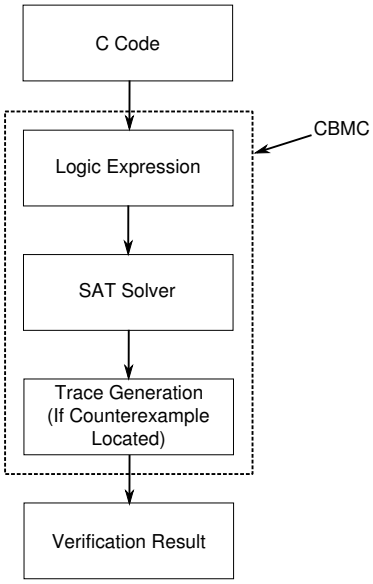


Figure 12.2: CBMC Processing Flow

Another alternative is to press the axioms of boolean logic into service, as described in the next section.

12.4 SAT Solvers

> Live by the heuristic, die by the heuristic.
>
> *Unknown*

Any finite program with bounded loops and recursion can be converted into a logic expression, which might express that program's assertions in terms of its inputs. Given such a logic expression, it would be quite interesting to know whether any possible combinations of inputs could result in one of the assertions triggering. If the inputs are expressed as combinations of boolean variables, this is simply SAT, also known as the satisfiability problem. SAT solvers are heavily used in verification of hardware, which has motivated great advances. A world-class early 1990s SAT solver might be able to handle a logic expression with 100 distinct boolean variables, but by the early 2010s million-variable SAT solvers were readily available [KS08].

In addition, front-end programs for SAT solvers can automatically translate C code into logic expressions, taking assertions into account and generating assertions for error conditions such as array-bounds errors. One example is the C bounded model checker, or `cbmc`, which is available as part of many Linux distributions. This tool is quite easy to use, with `cbmc test.c` sufficing to validate `test.c`, resulting in the processing flow shown in Figure 12.2. This ease of use is exceedingly important because it opens the door to formal verification being incorporated into regression-testing frameworks. In contrast, the traditional tools that require non-trivial translation to a special-purpose language are confined to design-time verification.

More recently, SAT solvers have appeared that handle parallel code. These solvers operate by converting the input code into single static assignment (SSA) form, then generating all permitted access orders. This approach seems promising, but it remains to be seen how well it works in practice. One encouraging sign is work in 2016 applying `cbmc` to Linux-kernel RCU [LMKM16, LMKM18, Roy17]. This work used minimal configurations of RCU, and verified scenarios using small numbers of threads, but nevertheless successfully ingested Linux-kernel C code and produced a useful result. The logic expressions generated from the C code had up to 90 million variables, 450 million clauses, occupied tens of gigabytes of memory, and required up to 80 hours of CPU time for the SAT solver to produce the correct result.

Nevertheless, a Linux-kernel hacker might be justified in feeling skeptical of a claim that his or her code had been automatically verified, and such hackers would find many fellow skeptics going back decades [DMLP79]. One way to productively express such skepticism is to provide bug-injected versions of the allegedly verified code. If the formal-verification tool finds all the injected bugs, our hacker might gain more confidence in the tool's capabilities. Of course, tools that find valid bugs of which the hacker was not yet aware will likely engender even more confidence. And this is exactly why there is a `git` archive with a 20-branch set of mutations, with each branch potentially containing a bug injected into Linux-kernel RCU [McK17]. Anyone with a formal-verification tool is cordially invited to try that tool out on this set of verification challenges.

Currently, `cbmc` is able to find a number of injected bugs, however, it has not yet been able to locate a bug that RCU's maintainer was not already aware of. Nevertheless, there is some reason to hope that SAT solvers will someday be useful for finding concurrency bugs in parallel code.

www.dbooks.org

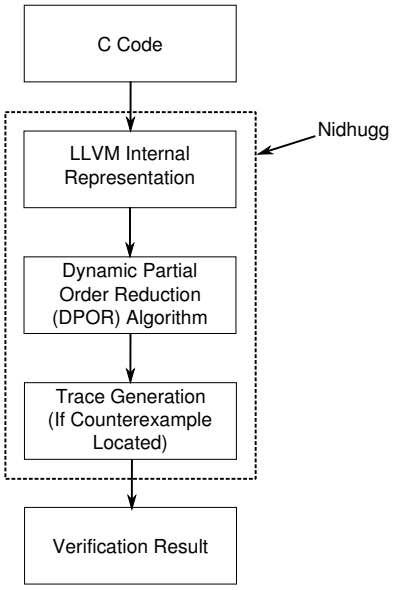


Figure 12.3: Nidhugg Processing Flow

12.5 Stateless Model Checkers

> He's making a list, he's permuting it twice. . .
>
> *with apologies to Haven Gillespie and J. Fred Coots*

The SAT-solver approaches described in the previous section are quite convenient and powerful, but the full tracking of all possible executions, including state, can incur substantial overhead. In fact, the memory and CPU-time overheads can sharply limit the size of programs that can be feasibly verified, which raises the question of whether less-exact approaches might find bugs in larger programs.

Although the jury is still out on this question, stateless model checkers such as Nidhugg [LSLK14] have in some cases handled larger programs [KS17b], and with similar ease of use, as illustrated by Figure 12.3. In addition, Nidhugg was more than an order of magnitude faster than was `cbmc` for some Linux-kernel RCU verification scenarios. Of course, Nidhugg's speed and scalability advantages are tied to the fact that it does not handle data non-determinism, but this was not a factor in these particular verification scenarios.

Nevertheless, as with `cbmc`, Nidhugg has not yet been able to locate a bug that Linux-kernel RCU's maintainer was not already aware of. However, it was able to demonstrate that one historical bug in Linux-kernel RCU was fixed by a different commit than the maintainer thought, which gives some additional hope that stateless model checkers like Nidhugg might someday be useful for finding concurrency bugs in parallel code.

12.6 Summary

> Western thought has focused on True-False; it is high time to shift to Robust-Fragile.
>
> *Nassim Nicholas Taleb, summarized*

The formal-verification techniques described in this chapter are very powerful tools for validating small parallel algorithms, but they should not be the only tools in your toolbox. Despite decades of focus on formal verification, testing remains the validation workhorse for large parallel software systems [Cor06a, Jon11, McK15c].

It is nevertheless quite possible that this will not always be the case. To see this, consider that there is estimated to be more than twenty billion instances of the Linux kernel as of 2017. Suppose that the Linux kernel has a bug that manifests on average every million years of runtime. As noted at the end of the preceding chapter, this bug will be appearing more than 50 times *per day* across the installed base. But the fact remains that most formal validation techniques can be used only on very small codebases. So what is a concurrency coder to do?

Think in terms of finding the first bug, the first relevant bug, the last relevant bug, and the last bug.

The first bug is normally found via inspection or compiler diagnostics. Although the increasingly sophisticated compiler diagnostics comprise a lightweight sort of formal verification, it is not common to think of them in those terms. This is in part due to an odd practitioner prejudice which says "If I am using it, it cannot be formal verification" on the one hand, and a large gap between compiler diagnostics and verification research on the other.

Although the first relevant bug might be located via inspection or compiler diagnostics, it is not unusual for these two steps to find only typos and false positives. Either way, the bulk of the relevant bugs, that is, those bugs that might actually be encountered in production, will often be found via testing.

When testing is driven by anticipated or real use cases, it is not uncommon for the last relevant bug to be located by testing. This situation might motivate a complete rejection of formal verification, however, irrelevant bugs

have an annoying habit of suddenly becoming relevant at the least convenient moment possible, courtesy of black-hat attacks. For security-critical software, which appears to be a continually increasing fraction of the total, there can thus be strong motivation to find and fix the last bug. Testing is demonstrably unable to find the last bug, so there is a possible role for formal verification, assuming, that is, that formal verification proves capable of growing into that role. As this chapter has shown, current formal verification systems are extremely limited.

Quick Quiz 12.35: But shouldn't sufficiently low-level software be for all intents and purposes immune to being exploited by black hats? ■

Please note that formal verification is often much harder to use than is testing. This is in part a cultural statement, and there is reason to hope that formal verification will be perceived to be easier with increased familiarity. That said, very simple test harnesses can find significant bugs in arbitrarily large software systems. In contrast, the effort required to apply formal verification seems to increase dramatically as the system size increases.

I have nevertheless made occasional use of formal verification for almost 30 years by playing to formal verification's strengths, namely design-time verification of small complex portions of the overarching software construct. The larger overarching software construct is of course validated by testing.

Quick Quiz 12.36: In light of the full verification of the L4 microkernel, isn't this limited view of formal verification just a little bit obsolete? ■

One final approach is to consider the following two definitions from Section 11.1.2 and the consequence that they imply:

Definition: Bug-free programs are trivial programs.

Definition: Reliable programs have no known bugs.

Consequence: Any non-trivial reliable program contains at least one as-yet-unknown bug.

From this viewpoint, any advances in validation and verification can have but two effects: (1) An increase in the number of trivial programs or (2) A decrease in the number of reliable programs. Of course, the human race's increasing reliance on multicore systems and software provides extreme motivation for a very sharp increase in the number of trivial programs.

However, if your code is so complex that you find yourself relying too heavily on formal-verification tools, you should carefully rethink your design, especially if your formal-verification tools require your code to be hand-translated to a special-purpose language. For example, a complex implementation of the dynticks interface for preemptible RCU that was presented in Section 12.1.5 turned out to have a much simpler alternative implementation, as discussed in Section 12.1.6.9. All else being equal, a simpler implementation is much better than a proof of correctness for a complex implementation.

And the open challenge to those working on formal verification techniques and systems is to prove this summary wrong! To assist in this task, Verification Challenge 6 is now available [McK17]. Have at it!!!

12.7 Choosing a Validation Plan

> Science is a first-rate piece of furniture for one's upper chamber, but only given common sense on the ground floor.
>
> *Oliver Wendell Holmes, updated*

What sort of validation should you use for your project?

As is often the case in software in particular and in engineering in general, the answer is "it depends".

Note that neither running a test nor undertaking formal verification will change your project. At best, such effort have an indirect effect by locating a bug that is later fixed. Nevertheless, fixing a bug might prevent inconvenience, monetary loss, property damage, or even loss of life. Clearly, this sort of indirect effect can be extremely valuable.

Unfortunately, as we have seen, it is difficult to predict whether or not a given validation effort will find important bugs. It is therefore all too easy to invest too little—or even to fail to invest at all, especially if development estimates proved overly optimistic or budgets unexpectedly tight, conditions which almost always come into play in real-world software projects.

The decision to nevertheless invest in validation is often forced by experienced people with forceful personalities. But this is no guarantee, given that other stakeholders might also have forceful personalities. Worse yet, these other stakeholders might bring stories of expensive validation efforts that nevertheless allowed embarrassing bugs to escape to the end users. So although a scarred, grey-haired, and grouchy veteran might carry the day, a more organized approach would perhaps be more useful.

www.dbooks.org

Fortunately, there is a strictly financial analog to investments in validation, and that is the insurance policy.

Both insurance policies and validation efforts require consistent up-front investments, and both defend against disasters that might or might not ever happen. Furthermore, both have exclusions of various types. For example, insurance policies for coastal areas might exclude damages due to tidal waves, while on the other hand we have seen that there is not yet any validation methodology that can find each and every bug.

In addition, it is possible to over-invest in both insurance and in validation. For but one example, a validation plan that consumed the entire development budget would be just as pointless as would an insurance policy that covered the Sun going nova.

One approach is to devote a given fraction of the software budget to validation, with that fraction depending on the criticality of the software, so that safety-critical avionics software might grant a larger fraction of its budget to validation than would a homework assignment. Where available, experience from prior similar projects should be brought to bear. However, it is necessary to structure the project so that the validation investment starts when the project does, otherwise the inevitable overruns in spending on coding will crowd out the validation effort.

Staffing start-up projects with experienced people can result in overinvestment in validation efforts. Just as it is possible to go broke buying too much insurance, it is possible to kill a project by investing too much in testing. This is especially the case for first-of-a-kind projects where it is not yet clear which use cases will be important, in which case testing for all possible use cases will be a possibly fatal waste of time, energy, and funding.

However, as the tasks supported by a start-up project become more routine, users often become less forgiving of failures, thus increasing the need for validation. Managing this shift in investment can be extremely challenging, especially in the all-too-common case where the users are unwilling or unable to disclose the exact nature of their use case. It then becomes critically important to reverse-engineer the use cases from bug reports and from discussions with the users. As these use cases are better understood, use of continuous integration can help reduce the cost of finding and fixing any bugs located.

One example evolution of a software project's use of validation is shown in Figure 12.4. As can be seen in the figure, Linux-kernel RCU didn't have any validation code whatsoever until Linux kernel v2.6.15, which was released more than two years after RCU was accepted into the kernel. The test suite achieved its peak fraction of the total lines of code in Linux kernel v2.6.19–v2.6.21. This fraction decreased sharply with the acceptance of preemptible RCU for real-time applications in v2.6.25. This decrease was due to the fact that the RCU API was identical in the preemptible and non-preemptible variants of RCU. This in turn meant that the existing test suite applied to both variants, so that even though the Linux-kernel RCU code expanded significantly, there was no need to expand the tests.

Subsequent bars in Figure 12.4 show that the RCU code base expanded significantly, but that the corresponding validation code expanded even more dramatically. Linux kernel v3.5 added tests for the `rcu_barrier()` API, closing a long-standing hole in test coverage. Linux kernel v3.14 added automated testing and analysis of test results, moving RCU towards continuous integration. Linux kernel v4.7 added a performance validation suite for RCU's update-side primitives. Linux kernel v4.12 added Tree SRCU, featuring improved update-side scalability, and v4.13 removed the old less-scalable SRCU implementation. Linux kernel v5.0 briefly hosted the `nolibc` library within the rcutorture scripting directory before it moved to its long-term home in `tools/include/nolibc`. Linux kernel v5.8 added the Tasks Trace and Rude flavors of RCU. Linux kernel v5.9 added the `refscale.c` suite of read-side performance tests. Numerous other changes may be found in the Linux kernel's `git` archives.

We have established that the validation budget varies from one project to the next, and also over the lifetime of any given project. But how should the validation investment be split between testing and formal verification?

This question is being answered naturally as compilers adopt increasingly aggressive formal-verification techniques into their diagnostics and as formal-verification tools continue to mature. In addition, the Linux-kernel lockdep and KCSAN tools illustrate the advantages of combining formal verification techniques with run-time analysis, as discussed in Section 11.3. Other combined techniques analyze traces gathered from executions [dOCdO19]. For the time being, the best practice is to focus first on testing and to reserve explicit work on formal verification for those portions of the project that are not well-served by testing, and that have exceptional needs for robustness. For example, Linux-kernel RCU relies primarily on testing, but has made occasional use of formal verification as discussed in this chapter.

In short, choosing a validation plan for concurrent software remains more an art than a science, let alone a

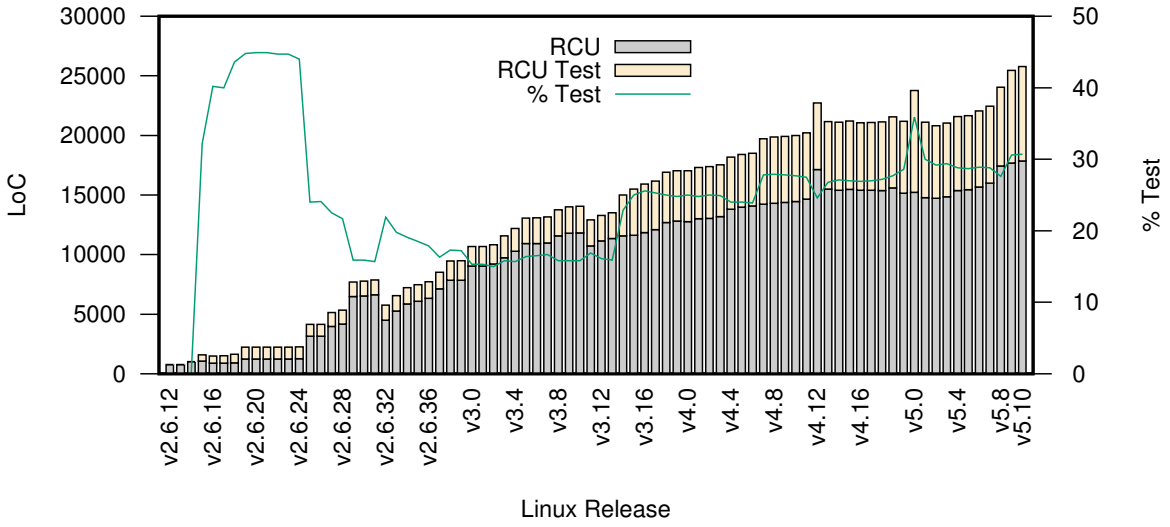


Figure 12.4: Linux-Kernel RCU Test Code

field of engineering. However, there is every reason to expect that increasingly rigorous approaches will continue to become more prevalent.

www.dbooks.org

You don't learn how to shoot and then learn how to launch and then learn to do a controlled spin—you learn to launch-shoot-spin.

"Ender's Shadow", Orson Scott Card

Chapter 13

Putting It All Together

This chapter gives some hints on concurrent-programming puzzles. Section 13.1 considers counter conundrums, Section 13.2 refurbishes reference counting, Section 13.3 helps with hazard pointers, Section 13.4 surmises on sequence-locking specials, and finally Section 13.5 reflects on RCU rescues.

13.1 Counter Conundrums

Ford carried on counting quietly. This is about the most aggressive thing you can do to a computer, the equivalent of going up to a human being and saying "Blood . . . blood . . . blood . . . blood . . ."

Douglas Adams

This section outlines solutions to counter conundrums.

13.1.1 Counting Updates

Suppose that Schrödinger (see Section 10.1) wants to count the number of updates for each animal, and that these updates are synchronized using a per-data-element lock. How can this counting best be done?

Of course, any number of counting algorithms from Chapter 5 might qualify, but the optimal approach is quite simple. Just place a counter in each data element, and increment it under the protection of that element's lock!

If readers access the count locklessly, then updaters should use `WRITE_ONCE()` to update the counter and lockless readers should use `READ_ONCE()` to load it.

13.1.2 Counting Lookups

Suppose that Schrödinger also wants to count the number of lookups for each animal, where lookups are protected by RCU. How can this counting best be done?

One approach would be to protect a lookup counter with the per-element lock, as discussed in Section 13.1.1. Unfortunately, this would require all lookups to acquire this lock, which would be a severe bottleneck on large systems.

Another approach is to "just say no" to counting, following the example of the `noatime` mount option. If this approach is feasible, it is clearly the best: After all, nothing is faster than doing nothing. If the lookup count cannot be dispensed with, read on!

Any of the counters from Chapter 5 could be pressed into service, with the statistical counters described in Section 5.2 being perhaps the most common choice. However, this results in a large memory footprint: The number of counters required is the number of data elements multiplied by the number of threads.

If this memory overhead is excessive, then one approach is to keep per-core or even per-socket counters rather than per-CPU counters, with an eye to the hash-table performance results depicted in Figure 10.3. This will require that the counter increments be atomic operations, especially for user-mode execution where a given thread could migrate to another CPU at any time.

If some elements are looked up very frequently, there are a number of approaches that batch updates by maintaining a per-thread log, where multiple log entries for a given element can be merged. After a given log entry has a sufficiently large increment or after sufficient time has passed, the log entries may be applied to the corresponding data elements. Silas Boyd-Wickizer has done some work formalizing this notion [BW14].

Edition.2-rc11

www.dbooks.org

13.2 Refurbish Reference Counting

> Counting is the religion of this generation. It is its hope and its salvation.
>
> *Gertrude Stein*

Although reference counting is a conceptually simple technique, many devils hide in the details when it is applied to concurrent software. After all, if the object was not subject to premature disposal, there would be no need for the reference counter in the first place. But if the object can be disposed of, what prevents disposal during the reference-acquisition process itself?

There are a number of ways to refurbish reference counters for use in concurrent software, including:

1. A lock residing outside of the object must be held while manipulating the reference count.
2. The object is created with a non-zero reference count, and new references may be acquired only when the current value of the reference counter is non-zero. If a thread does not have a reference to a given object, it might seek help from another thread that already has a reference.
3. In some cases, hazard pointers may be used as a drop-in replacement for reference counters.
4. An existence guarantee is provided for the object, thus preventing it from being freed while some other entity might be attempting to acquire a reference. Existence guarantees are often provided by automatic garbage collectors, and, as is seen in Sections 9.3 and 9.5, by hazard pointers and RCU, respectively.
5. A type-safety guarantee is provided for the object. An additional identity check must be performed once the reference is acquired. Type-safety guarantees can be provided by special-purpose memory allocators, for example, by the `SLAB_TYPESAFE_BY_RCU` feature within the Linux kernel, as is seen in Section 9.5.

Of course, any mechanism that provides existence guarantees by definition also provides type-safety guarantees. This results in four general categories of reference-acquisition protection: Reference counting, hazard pointers, sequence locking, and RCU.

Quick Quiz 13.1: Why not implement reference-acquisition using a simple compare-and-swap operation that only acquires a reference if the reference counter is non-zero? ■

Table 13.1: Synchronizing Reference Counting

Acquisition	Release			
	Locks	Reference Counts	Hazard Pointers	RCU
Locks	–	CAM	M	CA
Reference Counts	A	AM	M	A
Hazard Pointers	M	M	M	M
RCU	CA	MCA	M	CA

Given that the key reference-counting issue is synchronization between acquisition of a reference and freeing of the object, we have nine possible combinations of mechanisms, as shown in Table 13.1. This table divides reference-counting mechanisms into the following broad categories:

1. Simple counting with neither atomic operations, memory barriers, nor alignment constraints (“–”).
2. Atomic counting without memory barriers (“A”).
3. Atomic counting, with memory barriers required only on release (“AM”).
4. Atomic counting with a check combined with the atomic acquisition operation, and with memory barriers required only on release (“CAM”).
5. Atomic counting with a check combined with the atomic acquisition operation (“CA”).
6. Simple counting with a check combined with full memory barriers (“M”).
7. Atomic counting with a check combined with the atomic acquisition operation, and with memory barriers also required on acquisition (“MCA”).

However, because all Linux-kernel atomic operations that return a value are defined to contain memory barriers,[1] all release operations contain memory barriers, and all checked acquisition operations also contain memory barriers. Therefore, cases “CA” and “MCA” are equivalent to “CAM”, so that there are sections below for only the first four cases and the sixth case: “–”, “A”, “AM”, “CAM”, and “M”. Later sections describe optimizations that can

[1] With `atomic_read()` and `ATOMIC_INIT()` being the exceptions that prove the rule.

improve performance if reference acquisition and release is very frequent, and the reference count need be checked for zero only very rarely.

13.2.1 Implementation of Reference-Counting Categories

Simple counting protected by locking ("–") is described in Section 13.2.1.1, atomic counting with no memory barriers ("A") is described in Section 13.2.1.2, atomic counting with acquisition memory barrier ("AM") is described in Section 13.2.1.3, and atomic counting with check and release memory barrier ("CAM") is described in Section 13.2.1.4. Use of hazard pointers is described in Section 9.3 on page 9.3 and in Section 13.3.

13.2.1.1 Simple Counting

Simple counting, with neither atomic operations nor memory barriers, can be used when the reference-counter acquisition and release are both protected by the same lock. In this case, it should be clear that the reference count itself may be manipulated non-atomically, because the lock provides any necessary exclusion, memory barriers, atomic instructions, and disabling of compiler optimizations. This is the method of choice when the lock is required to protect other operations in addition to the reference count, but where a reference to the object must be held after the lock is released. Listing 13.1 shows a simple API that might be used to implement simple non-atomic reference counting—although simple reference counting is almost always open-coded instead.

Listing 13.1: Simple Reference-Count API

```
1  struct sref {
2    int refcount;
3  };
4
5  void sref_init(struct sref *sref)
6  {
7    sref->refcount = 1;
8  }
9
10 void sref_get(struct sref *sref)
11 {
12   sref->refcount++;
13 }
14
15 int sref_put(struct sref *sref,
16              void (*release)(struct sref *sref))
17 {
18   WARN_ON(release == NULL);
19   WARN_ON(release == (void (*)(struct sref *))kfree);
20
21   if (--sref->refcount == 0) {
22     release(sref);
23     return 1;
24   }
25   return 0;
26 }
```

13.2.1.2 Atomic Counting

Simple atomic counting may be used in cases where any CPU acquiring a reference must already hold a reference. This style is used when a single CPU creates an object for its own private use, but must allow for accesses from other CPUs, tasks, timer handlers, and so on. Any CPU that hands the object off must first acquire a new reference on behalf of the recipient on the one hand, or refrain from further accesses after the handoff on the other. In the Linux kernel, the `kref` primitives are used to implement this style of reference counting, as shown in Listing 13.2.[2]

Atomic counting is required in this case because locking does not protect all reference-count operations, which means that two different CPUs might concurrently manipulate the reference count. If normal increment and decrement were used, a pair of CPUs might both fetch the reference count concurrently, perhaps both obtaining the value "3". If both of them increment their value, they will both obtain "4", and both will store this value back into the counter. Since the new value of the counter should instead be "5", one of the increments has been lost. Therefore, atomic operations must be used both for counter increments and for counter decrements.

If releases are guarded by locking, hazard pointers, or RCU, memory barriers are *not* required, but for different reasons. In the case of locking, the locks provide any needed memory barriers (and disabling of compiler optimizations), and the locks also prevent a pair of releases from running concurrently. In the case of hazard pointers and RCU, cleanup must be deferred until all currently executing RCU read-side critical sections have completed, and any needed memory barriers or disabling of compiler optimizations will be provided by the RCU infrastructure. Therefore, if two CPUs release the final two references concurrently, the actual cleanup will be deferred until both CPUs exit their RCU read-side critical sections.

Quick Quiz 13.2: Why isn't it necessary to guard against cases where one CPU acquires a reference just after another CPU releases the last reference? ■

The `kref` structure itself, consisting of a single atomic data item, is shown in lines 1–3 of Listing 13.2. The `kref_`

[2] As of Linux v4.10. Linux v4.11 introduced a `refcount_t` API that improves efficiency weakly ordered platforms, but which is functionally equivalent to the `atomic_t` that it replaced.

www.dbooks.org

Listing 13.2: Linux Kernel `kref` API

```
1 struct kref {
2   atomic_t refcount;
3 };
4
5 void kref_init(struct kref *kref)
6 {
7   atomic_set(&kref->refcount, 1);
8 }
9
10 void kref_get(struct kref *kref)
11 {
12   WARN_ON(!atomic_read(&kref->refcount));
13   atomic_inc(&kref->refcount);
14 }
15
16 static inline int
17 kref_sub(struct kref *kref, unsigned int count,
18          void (*release)(struct kref *kref))
19 {
20   WARN_ON(release == NULL);
21
22   if (atomic_sub_and_test((int) count,
23                           &kref->refcount)) {
24     release(kref);
25     return 1;
26   }
27   return 0;
28 }
```

`init()` function on lines 5–8 initializes the counter to the value "1". Note that the `atomic_set()` primitive is a simple assignment, the name stems from the data type of `atomic_t` rather than from the operation. The `kref_init()` function must be invoked during object creation, before the object has been made available to any other CPU.

The `kref_get()` function on lines 10–14 unconditionally atomically increments the counter. The `atomic_inc()` primitive does not necessarily explicitly disable compiler optimizations on all platforms, but the fact that the `kref` primitives are in a separate module and that the Linux kernel build process does no cross-module optimizations has the same effect.

The `kref_sub()` function on lines 16–28 atomically decrements the counter, and if the result is zero, line 24 invokes the specified `release()` function and line 25 returns, informing the caller that `release()` was invoked. Otherwise, `kref_sub()` returns zero, informing the caller that `release()` was not called.

Quick Quiz 13.3: Suppose that just after the `atomic_sub_and_test()` on line 22 of Listing 13.2 is invoked, that some other CPU invokes `kref_get()`. Doesn't this result in that other CPU now having an illegal reference to a released object? ■

Listing 13.3: Linux Kernel `dst_clone` API

```
1 static inline
2 struct dst_entry * dst_clone(struct dst_entry * dst)
3 {
4   if (dst)
5     atomic_inc(&dst->__refcnt);
6   return dst;
7 }
8
9 static inline
10 void dst_release(struct dst_entry * dst)
11 {
12   if (dst) {
13     WARN_ON(atomic_read(&dst->__refcnt) < 1);
14     smp_mb__before_atomic_dec();
15     atomic_dec(&dst->__refcnt);
16   }
17 }
```

Quick Quiz 13.4: Suppose that `kref_sub()` returns zero, indicating that the `release()` function was not invoked. Under what conditions can the caller rely on the continued existence of the enclosing object? ■

Quick Quiz 13.5: Why not just pass `kfree()` as the release function? ■

13.2.1.3 Atomic Counting With Release Memory Barrier

Atomic reference counting with release memory barriers is used by the Linux kernel's networking layer to track the destination caches that are used in packet routing. The actual implementation is quite a bit more involved; this section focuses on the aspects of `struct dst_entry` reference-count handling that matches this use case, shown in Listing 13.3.[3]

The `dst_clone()` primitive may be used if the caller already has a reference to the specified `dst_entry`, in which case it obtains another reference that may be handed off to some other entity within the kernel. Because a reference is already held by the caller, `dst_clone()` need not execute any memory barriers. The act of handing the `dst_entry` to some other entity might or might not require a memory barrier, but if such a memory barrier is required, it will be embedded in the mechanism used to hand the `dst_entry` off.

The `dst_release()` primitive may be invoked from any environment, and the caller might well reference elements of the `dst_entry` structure immediately prior to the call to `dst_release()`. The `dst_release()` primitive

[3] As of Linux v4.13. Linux v4.14 added a level of indirection to permit more comprehensive debugging checks, but the overall effect in the absence of bugs is identical.

therefore contains a memory barrier on line 14 preventing both the compiler and the CPU from misordering accesses.

Please note that the programmer making use of `dst_clone()` and `dst_release()` need not be aware of the memory barriers, only of the rules for using these two primitives.

13.2.1.4 Atomic Counting With Check and Release Memory Barrier

Consider a situation where the caller must be able to acquire a new reference to an object to which it does not already hold a reference, but where that object's existence is guaranteed. The fact that initial reference-count acquisition can now run concurrently with reference-count release adds further complications. Suppose that a reference-count release finds that the new value of the reference count is zero, signaling that it is now safe to clean up the reference-counted object. We clearly cannot allow a reference-count acquisition to start after such clean-up has commenced, so the acquisition must include a check for a zero reference count. This check must be part of the atomic increment operation, as shown below.

Quick Quiz 13.6: Why can't the check for a zero reference count be made in a simple "if" statement with an atomic increment in its "then" clause? ■

The Linux kernel's `fget()` and `fput()` primitives use this style of reference counting. Simplified versions of these functions are shown in Listing 13.4.[4]

Line 4 of `fget()` fetches the pointer to the current process's file-descriptor table, which might well be shared with other processes. Line 6 invokes `rcu_read_lock()`, which enters an RCU read-side critical section. The callback function from any subsequent `call_rcu()` primitive will be deferred until a matching `rcu_read_unlock()` is reached (line 10 or 14 in this example). Line 7 looks up the file structure corresponding to the file descriptor specified by the `fd` argument, as will be described later. If there is an open file corresponding to the specified file descriptor, then line 9 attempts to atomically acquire a reference count. If it fails to do so, lines 10–11 exit the RCU read-side critical section and report failure. Otherwise, if the attempt is successful, lines 14–15 exit the read-side critical section and return a pointer to the file structure.

The `fcheck_files()` primitive is a helper function for `fget()`. Line 22 uses `rcu_dereference()` to safely fetch an RCU-protected pointer to this task's current file-descriptor table, and line 24 checks to see if the specified file descriptor is in range. If so, line 25 fetches the pointer to the file structure, again using the `rcu_dereference()` primitive. Line 26 then returns a pointer to the file structure or `NULL` in case of failure.

Listing 13.4: Linux Kernel `fget/fput` API

```
1  struct file *fget(unsigned int fd)
2  {
3    struct file *file;
4    struct files_struct *files = current->files;
5
6    rcu_read_lock();
7    file = fcheck_files(files, fd);
8    if (file) {
9      if (!atomic_inc_not_zero(&file->f_count)) {
10       rcu_read_unlock();
11       return NULL;
12     }
13   }
14   rcu_read_unlock();
15   return file;
16 }
17
18 struct file *
19 fcheck_files(struct files_struct *files, unsigned int fd)
20 {
21   struct file * file = NULL;
22   struct fdtable *fdt = rcu_dereference((files)->fdt);
23
24   if (fd < fdt->max_fds)
25     file = rcu_dereference(fdt->fd[fd]);
26   return file;
27 }
28
29 void fput(struct file *file)
30 {
31   if (atomic_dec_and_test(&file->f_count))
32     call_rcu(&file->f_u.fu_rcuhead, file_free_rcu);
33 }
34
35 static void file_free_rcu(struct rcu_head *head)
36 {
37   struct file *f;
38
39   f = container_of(head, struct file, f_u.fu_rcuhead);
40   kmem_cache_free(filp_cachep, f);
41 }
```

The `fput()` primitive releases a reference to a file structure. Line 31 atomically decrements the reference count, and, if the result was zero, line 32 invokes the `call_rcu()` primitives in order to free up the file structure (via the `file_free_rcu()` function specified in `call_rcu()`'s second argument), but only after all currently-executing RCU read-side critical sections complete, that is, after an RCU grace period has elapsed.

Once the grace period completes, the `file_free_rcu()` function obtains a pointer to the file structure on line 39, and frees it on line 40.

[4] As of Linux v2.6.38. Additional `O_PATH` functionality was added in v2.6.39, refactoring was applied in v3.14, and `mmap_sem` contention was reduced in v4.1.

www.dbooks.org

This code fragment thus demonstrates how RCU can be used to guarantee existence while an in-object reference count is being incremented.

13.2.2 Counter Optimizations

In some cases where increments and decrements are common, but checks for zero are rare, it makes sense to maintain per-CPU or per-task counters, as was discussed in Chapter 5. For example, see the paper on sleepable read-copy update (SRCU), which applies this technique to RCU [McK06]. This approach eliminates the need for atomic instructions or memory barriers on the increment and decrement primitives, but still requires that code-motion compiler optimizations be disabled. In addition, the primitives such as `synchronize_srcu()` that check for the aggregate reference count reaching zero can be quite slow. This underscores the fact that these techniques are designed for situations where the references are frequently acquired and released, but where it is rarely necessary to check for a zero reference count.

However, it is usually the case that use of reference counts requires writing (often atomically) to a data structure that is otherwise read only. In this case, reference counts are imposing expensive cache misses on readers.

It is therefore worthwhile to look into synchronization mechanisms that do not require readers to write to the data structure being traversed. One possibility is the hazard pointers covered in Section 9.3 and another is RCU, which is covered in Section 9.5.

13.3 Hazard-Pointer Helpers

> It's the little things that count, hundreds of them.
>
> *Cliff Shaw*

This section looks at some issues that can arise when dealing with hash tables. Please note that these issues also apply to many other search structures.

13.3.1 Scalable Reference Count

Suppose a reference count is becoming a performance or scalability bottleneck. What can you do?

One approach is to instead use hazard pointers.

There are some differences, perhaps most notably that with hazard pointers it is extremely expensive to determine when the corresponding reference count has reached zero.

One way to work around this problem is to split the load between reference counters and hazard pointers. Each data element has a reference counter that tracks the number of other data elements referencing this element on the one hand, and readers use hazard pointers on the other.

Making this arrangement work both efficiently and correctly can be quite challenging, and so interested readers are invited to examine the UnboundedQueue and ConcurrentHashMap data structures implemented in Folly open-source library.[5]

13.4 Sequence-Locking Specials

> The girl who can't dance says the band can't play.
>
> *Yiddish proverb*

This section looks at some special uses of sequence counters.

13.4.1 Correlated Data Elements

Suppose that have a hash table where we need correlated views of two or more of the elements. These elements are updated together, and we do not want to see an old version of the first element along with new versions of the other elements. For example, Schrödinger decided to add his extended family to his in-memory database along with all his animals. Although Schrödinger understands that marriages and divorces do not happen instantaneously, he is also a traditionalist. As such, he absolutely does not want his database ever to show that the bride is now married, but the groom is not, and vice versa. Plus, if you think Schrödinger is a traditionalist, you just try conversing with some of his family members! In other words, Schrödinger wants to be able to carry out a wedlock-consistent traversal of his database.

One approach is to use sequence locks (see Section 9.4), so that wedlock-related updates are carried out under the protection of `write_seqlock()`, while reads requiring wedlock consistency are carried out within a `read_seqbegin()` / `read_seqretry()` loop. Note that sequence locks are not a replacement for RCU protection: Sequence locks protect against concurrent modifications, but RCU is still needed to protect against concurrent deletions.

[5] `https://github.com/facebook/folly`

This approach works quite well when the number of correlated elements is small, the time to read these elements is short, and the update rate is low. Otherwise, updates might happen so quickly that readers might never complete. Although Schrödinger does not expect that even his least-sane relatives will marry and divorce quickly enough for this to be a problem, he does realize that this problem could well arise in other situations. One way to avoid this reader-starvation problem is to have the readers use the update-side primitives if there have been too many retries, but this can degrade both performance and scalability. Another way to avoid starvation is to have multiple sequence locks, in Schrödinger's case, perhaps one per species.

In addition, if the update-side primitives are used too frequently, poor performance and scalability will result due to lock contention. One way to avoid this is to maintain a per-element sequence lock, and to hold both spouses' locks when updating their marital status. Readers can do their retry looping on either of the spouses' locks to gain a stable view of any change in marital status involving both members of the pair. This avoids contention due to high marriage and divorce rates, but complicates gaining a stable view of all marital statuses during a single scan of the database.

If the element groupings are well-defined and persistent, which marital status is hoped to be, then one approach is to add pointers to the data elements to link together the members of a given group. Readers can then traverse these pointers to access all the data elements in the same group as the first one located.

This technique is used heavily in the Linux kernel, perhaps most notably in the dcache subsystem [Bro15b]. Note that it is likely that similar schemes also work with hazard pointers.

Another approach is to shard the data elements, and then have each update write-acquire all the sequence locks needed to cover the data elements affected by by that update. Of course, these write-acquisitions must be done carefully in order to avoid deadlock. Readers would also need to read-acquire multiple sequence locks, but in the surprisingly common case where readers only look up one data element, only one sequence lock need be read-acquired.

This approach provides sequential consistency to readers, each of which will either see the effects of a given update or not, with any partial updates resulting in a read-side retry. Sequential consistency is an extremely strong guarantee, incurring equally strong restrictions and equally high overheads. In this case, we saw that readers might be starved on the one hand, or might need to acquire the update-side lock on the other. Although this works very well in cases where updates are infrequent, it unnecessarily forces read-side retries even when the update did not affect any of the data that a retried reader accesses. Section 13.5.4 therefore covers a much weaker form of consistency that not only avoids reader starvation, but also avoids any form of read-side retry.

13.4.2 Upgrade to Writer

As discussed in Section 9.5.4.2, RCU permits readers to upgrade to writers. This capability can be quite useful when a reader scanning an RCU-protected data structure notices that the current element needs to be updated. What happens when you try this trick with sequence locking?

It turns out that this sequence-locking trick is actually used in the Linux kernel, for example, by the `sdma_flush()` function in `drivers/infiniband/hw/hfi1/sdma.c`. The effect is to doom the enclosing reader to retry. This trick is therefore used when the reader detects some condition that requires a retry.

13.5 RCU Rescues

> With great doubts comes great understanding, with little doubts comes little understanding.
>
> *Chinese proverb*

This section shows how to apply RCU to some examples discussed earlier in this book. In some cases, RCU provides simpler code, in other cases better performance and scalability, and in still other cases, both.

13.5.1 RCU and Per-Thread-Variable-Based Statistical Counters

Section 5.2.3 described an implementation of statistical counters that provided excellent performance, roughly that of simple increment (as in the C ++ operator), and linear scalability—but only for incrementing via `inc_count()`. Unfortunately, threads needing to read out the value via `read_count()` were required to acquire a global lock, and thus incurred high overhead and suffered poor scalability. The code for the lock-based implementation is shown in Listing 5.4 on Page 53.

www.dbooks.org

Quick Quiz 13.7: Why on earth did we need that global lock in the first place? ■

13.5.1.1 Design

The hope is to use RCU rather than `final_mutex` to protect the thread traversal in `read_count()` in order to obtain excellent performance and scalability from `read_count()`, rather than just from `inc_count()`. However, we do not want to give up any accuracy in the computed sum. In particular, when a given thread exits, we absolutely cannot lose the exiting thread's count, nor can we double-count it. Such an error could result in inaccuracies equal to the full precision of the result, in other words, such an error would make the result completely useless. And in fact, one of the purposes of `final_mutex` is to ensure that threads do not come and go in the middle of `read_count()` execution.

Therefore, if we are to dispense with `final_mutex`, we will need to come up with some other method for ensuring consistency. One approach is to place the total count for all previously exited threads and the array of pointers to the per-thread counters into a single structure. Such a structure, once made available to `read_count()`, is held constant, ensuring that `read_count()` sees consistent data.

13.5.1.2 Implementation

Lines 1–4 of Listing 13.5 show the `countarray` structure, which contains a `->total` field for the count from previously exited threads, and a `counterp[]` array of pointers to the per-thread `counter` for each currently running thread. This structure allows a given execution of `read_count()` to see a total that is consistent with the indicated set of running threads.

Lines 6–8 contain the definition of the per-thread `counter` variable, the global pointer `countarrayp` referencing the current `countarray` structure, and the `final_mutex` spinlock.

Lines 10–13 show `inc_count()`, which is unchanged from Listing 5.4.

Lines 15–31 show `read_count()`, which has changed significantly. Lines 22 and 29 substitute `rcu_read_lock()` and `rcu_read_unlock()` for acquisition and release of `final_mutex`. Line 23 uses `rcu_dereference()` to snapshot the current `countarray` structure into local variable `cap`. Proper use of RCU will guarantee that this `countarray` structure will remain with

Listing 13.5: RCU and Per-Thread Statistical Counters

```
1  struct countarray {
2    unsigned long total;
3    unsigned long *counterp[NR_THREADS];
4  };
5
6  unsigned long __thread counter = 0;
7  struct countarray *countarrayp = NULL;
8  DEFINE_SPINLOCK(final_mutex);
9
10 __inline__ void inc_count(void)
11 {
12   WRITE_ONCE(counter, counter + 1);
13 }
14
15 unsigned long read_count(void)
16 {
17   struct countarray *cap;
18   unsigned long *ctrp;
19   unsigned long sum;
20   int t;
21
22   rcu_read_lock();
23   cap = rcu_dereference(countarrayp);
24   sum = READ_ONCE(cap->total);
25   for_each_thread(t) {
26     ctrp = READ_ONCE(cap->counterp[t]);
27     if (ctrp != NULL) sum += *ctrp;
28   }
29   rcu_read_unlock();
30   return sum;
31 }
32
33 void count_init(void)
34 {
35   countarrayp = malloc(sizeof(*countarrayp));
36   if (countarrayp == NULL) {
37     fprintf(stderr, "Out of memory\n");
38     exit(EXIT_FAILURE);
39   }
40   memset(countarrayp, '\0', sizeof(*countarrayp));
41 }
42
43 void count_register_thread(unsigned long *p)
44 {
45   int idx = smp_thread_id();
46
47   spin_lock(&final_mutex);
48   countarrayp->counterp[idx] = &counter;
49   spin_unlock(&final_mutex);
50 }
51
52 void count_unregister_thread(int nthreadsexpected)
53 {
54   struct countarray *cap;
55   struct countarray *capold;
56   int idx = smp_thread_id();
57
58   cap = malloc(sizeof(*countarrayp));
59   if (cap == NULL) {
60     fprintf(stderr, "Out of memory\n");
61     exit(EXIT_FAILURE);
62   }
63   spin_lock(&final_mutex);
64   *cap = *countarrayp;
65   WRITE_ONCE(cap->total, cap->total + counter);
66   cap->counterp[idx] = NULL;
67   capold = countarrayp;
68   rcu_assign_pointer(countarrayp, cap);
69   spin_unlock(&final_mutex);
70   synchronize_rcu();
71   free(capold);
72 }
```

us through at least the end of the current RCU read-side critical section at line 29. Line 24 initializes `sum` to `cap->total`, which is the sum of the counts of threads that have previously exited. Lines 25–27 add up the per-thread counters corresponding to currently running threads, and, finally, line 30 returns the sum.

The initial value for `countarrayp` is provided by `count_init()` on lines 33–41. This function runs before the first thread is created, and its job is to allocate and zero the initial structure, and then assign it to `countarrayp`.

Lines 43–50 show the `count_register_thread()` function, which is invoked by each newly created thread. Line 45 picks up the current thread's index, line 47 acquires `final_mutex`, line 48 installs a pointer to this thread's `counter`, and line 49 releases `final_mutex`.

Quick Quiz 13.8: Hey!!! Line 48 of Listing 13.5 modifies a value in a pre-existing `countarray` structure! Didn't you say that this structure, once made available to `read_count()`, remained constant??? ■

Lines 52–72 show `count_unregister_thread()`, which is invoked by each thread just before it exits. Lines 58–62 allocate a new `countarray` structure, line 63 acquires `final_mutex` and line 69 releases it. Line 64 copies the contents of the current `countarray` into the newly allocated version, line 65 adds the exiting thread's `counter` to new structure's `->total`, and line 66 `NULL`s the exiting thread's `counterp[]` array element. Line 67 then retains a pointer to the current (soon to be old) `countarray` structure, and line 68 uses `rcu_assign_pointer()` to install the new version of the `countarray` structure. Line 70 waits for a grace period to elapse, so that any threads that might be concurrently executing in `read_count()`, and thus might have references to the old `countarray` structure, will be allowed to exit their RCU read-side critical sections, thus dropping any such references. Line 71 can then safely free the old `countarray` structure.

Quick Quiz 13.9: Given the fixed-size `counterp` array, exactly how does this code avoid a fixed upper bound on the number of threads??? ■

13.5.1.3 Discussion

Quick Quiz 13.10: Wow! Listing 13.5 contains 70 lines of code, compared to only 42 in Listing 5.4. Is this extra complexity really worth it? ■

Use of RCU enables exiting threads to wait until other threads are guaranteed to be done using the exiting threads' `__thread` variables. This allows the `read_count()` function to dispense with locking, thereby providing excellent performance and scalability for both the `inc_count()` and `read_count()` functions. However, this performance and scalability come at the cost of some increase in code complexity. It is hoped that compiler and library writers employ user-level RCU [Des09b] to provide safe cross-thread access to `__thread` variables, greatly reducing the complexity seen by users of `__thread` variables.

13.5.2 RCU and Counters for Removable I/O Devices

Section 5.4.6 showed a fanciful pair of code fragments for dealing with counting I/O accesses to removable devices. These code fragments suffered from high overhead on the fastpath (starting an I/O) due to the need to acquire a reader-writer lock.

This section shows how RCU may be used to avoid this overhead.

The code for performing an I/O is quite similar to the original, with an RCU read-side critical section being substituted for the reader-writer lock read-side critical section in the original:

```
1  rcu_read_lock();
2  if (removing) {
3          rcu_read_unlock();
4          cancel_io();
5  } else {
6          add_count(1);
7          rcu_read_unlock();
8          do_io();
9          sub_count(1);
10 }
```

The RCU read-side primitives have minimal overhead, thus speeding up the fastpath, as desired.

The updated code fragment removing a device is as follows:

```
1  spin_lock(&mylock);
2  removing = 1;
3  sub_count(mybias);
4  spin_unlock(&mylock);
5  synchronize_rcu();
6  while (read_count() != 0) {
7          poll(NULL, 0, 1);
8  }
9  remove_device();
```

Here we replace the reader-writer lock with an exclusive spinlock and add a `synchronize_rcu()` to wait for all of the RCU read-side critical sections to complete. Because

www.dbooks.org

Listing 13.6: RCU-Protected Variable-Length Array

```
1 struct foo {
2         int length;
3         char *a;
4 };
```

Listing 13.7: Improved RCU-Protected Variable-Length Array

```
1 struct foo_a {
2         int length;
3         char a[0];
4 };
5
6 struct foo {
7         struct foo_a *fa;
8 };
```

of the `synchronize_rcu()`, once we reach line 6, we know that all remaining I/Os have been accounted for.

Of course, the overhead of `synchronize_rcu()` can be large, but given that device removal is quite rare, this is usually a good tradeoff.

13.5.3 Array and Length

Suppose we have an RCU-protected variable-length array, as shown in Listing 13.6. The length of the array `->a[]` can change dynamically, and at any given time, its length is given by the field `->length`. Of course, this introduces the following race condition:

1. The array is initially 16 characters long, and thus `->length` is equal to 16.
2. CPU 0 loads the value of `->length`, obtaining the value 16.
3. CPU 1 shrinks the array to be of length 8, and assigns a pointer to a new 8-character block of memory into `->a[]`.
4. CPU 0 picks up the new pointer from `->a[]`, and stores a new value into element 12. Because the array has only 8 characters, this results in a SEGV or (worse yet) memory corruption.

How can we prevent this?

One approach is to make careful use of memory barriers, which are covered in Chapter 15. This works, but incurs read-side overhead and, perhaps worse, requires use of explicit memory barriers.

A better approach is to put the value and the array into the same structure, as shown in Listing 13.7 [ACMS03]. Allocating a new array (`foo_a` structure) then automatically provides a new place for the array length. This means that if any CPU picks up a reference to `->fa`, it is guaranteed that the `->length` will match the `->a[]`.

Listing 13.8: Uncorrelated Measurement Fields

```
1 struct animal {
2         char name[40];
3         double age;
4         double meas_1;
5         double meas_2;
6         double meas_3;
7         char photo[0]; /* large bitmap. */
8 };
```

1. The array is initially 16 characters long, and thus `->length` is equal to 16.
2. CPU 0 loads the value of `->fa`, obtaining a pointer to the structure containing the value 16 and the 16-byte array.
3. CPU 0 loads the value of `->fa->length`, obtaining the value 16.
4. CPU 1 shrinks the array to be of length 8, and assigns a pointer to a new `foo_a` structure containing an 8-character block of memory into `->fa`.
5. CPU 0 picks up the new pointer from `->a[]`, and stores a new value into element 12. But because CPU 0 is still referencing the old `foo_a` structure that contains the 16-byte array, all is well.

Of course, in both cases, CPU 1 must wait for a grace period before freeing the old array.

A more general version of this approach is presented in the next section.

13.5.4 Correlated Fields

Suppose that each of Schödinger's animals is represented by the data element shown in Listing 13.8. The `meas_1`, `meas_2`, and `meas_3` fields are a set of correlated measurements that are updated periodically. It is critically important that readers see these three values from a single measurement update: If a reader sees an old value of `meas_1` but new values of `meas_2` and `meas_3`, that reader will become fatally confused. How can we guarantee that readers will see coordinated sets of these three values?[6]

One approach would be to allocate a new `animal` structure, copy the old structure into the new structure,

[6] This situation is similar to that described in Section 13.4.1, except that readers need only see a consistent view of a given single data element, not a consistent view of a group of data elements.

Listing 13.9: Correlated Measurement Fields

```
1 struct measurement {
2         double meas_1;
3         double meas_2;
4         double meas_3;
5 };
6
7 struct animal {
8         char name[40];
9         double age;
10        struct measurement *mp;
11        char photo[0]; /* large bitmap. */
12 };
```

update the new structure's `meas_1`, `meas_2`, and `meas_3` fields, and then replace the old structure with a new one by updating the pointer. This does guarantee that all readers see coordinated sets of measurement values, but it requires copying a large structure due to the `->photo[]` field. This copying might incur unacceptably large overhead.

Another approach is to impose a level of indirection, as shown in Listing 13.9 [McK04, Section 5.3.4]. When a new measurement is taken, a new `measurement` structure is allocated, filled in with the measurements, and the `animal` structure's `->mp` field is updated to point to this new `measurement` structure using `rcu_assign_pointer()`. After a grace period elapses, the old `measurement` structure can be freed.

Quick Quiz 13.11: But cant't the approach shown in Listing 13.9 result in extra cache misses, in turn resulting in additional read-side overhead? ■

This approach enables readers to see correlated values for selected fields, but while incurring minimal read-side overhead. This per-data-element consistency suffices in the common case where a reader looks only at a single data element.

13.5.5 Update-Friendly Traversal

Suppose that a statistical scan of all elements in a hash table is required. For example, Schrödinger might wish to compute the average length-to-weight ratio over all of his animals.[7] Suppose further that Schrödinger is willing to ignore slight errors due to animals being added to and removed from the hash table while this statistical scan is being carried out. What should Schrödinger do to control concurrency?

One approach is to enclose the statistical scan in an RCU read-side critical section. This permits updates to proceed concurrently without unduly impeding the scan. In particular, the scan does not block the updates and vice versa, which allows scan of hash tables containing very large numbers of elements to be supported gracefully, even in the face of very high update rates.

Quick Quiz 13.12: But how does this scan work while a resizable hash table is being resized? In that case, neither the old nor the new hash table is guaranteed to contain all the elements in the hash table! ■

13.5.6 Scalable Reference Count Two

Suppose a reference count is becoming a performance or scalability bottleneck. What can you do?

Another approach is to use per-CPU counters for each reference count, somewhat similar to the algorithms in Chapter 5, in particular, the exact limit counters described in Section 5.4. The need to switch between per-CPU and global modes for these counters results either in expensive increments and decrements on the one hand (Section 5.4.1) or in the use of POSIX signals on the other (Section 5.4.3).

Another alternative is to use RCU to mediate the switch between per-CPU and global counting modes. Each update is carried out within an RCU read-side critical section, and each update checks a flag to determine whether to update the per-CPU counters on the one hand or the global on the other. To switch modes, update the flag, wait for a grace period, and then move any remaining counts from the per-CPU counters to the global counter or vice versa.

The Linux kernel uses this approach in its `percpu_ref` style of reference counter, to which interested readers are referred.

[7] Why would such a quantity be useful? Beats me! But group statistics are often useful.

www.dbooks.org

If a little knowledge is a dangerous thing, just think what you could do with a lot of knowledge!

Unknown

Chapter 14

Advanced Synchronization

This chapter covers synchronization techniques used for lockless algorithms and parallel real-time systems.

Although lockless algorithms can be quite helpful when faced with extreme requirements, they are no panacea. For example, as noted at the end of Chapter 5, you should thoroughly apply partitioning, batching, and well-tested packaged weak APIs (see Chapters 8 and 9) before even thinking about lockless algorithms.

But after doing all that, you still might find yourself needing the advanced techniques described in this chapter. To that end, Section 14.1 summarizes techniques used thus far for avoiding locks and Section 14.2 gives a brief overview of non-blocking synchronization. Memory ordering is also quite important, but it warrants its own chapter, namely Chapter 15.

The second form of advanced synchronization provides the stronger forward-progress guarantees needed for parallel real-time computing, which is the topic of Section 14.3.

14.1 Avoiding Locks

We are confronted with insurmountable opportunities.

Walt Kelly

Although locking is the workhorse of parallelism in production, in many situations performance, scalability, and real-time response can all be greatly improved through use of lockless techniques. A particularly impressive example of such a lockless technique are the statistical counters described in Section 5.2, which avoids not only locks, but also atomic operations, memory barriers, and even cache misses for counter increments. Other examples we have covered include:

1. The fastpaths through a number of other counting algorithms in Chapter 5.
2. The fastpath through resource allocator caches in Section 6.4.3.
3. The maze solver in Section 6.5.
4. The data-ownership techniques in Chapter 8.
5. The reference-counting, hazard-pointer, and RCU techniques in Chapter 9.
6. The lookup code paths in Chapter 10.
7. Many of the techniques in Chapter 13.

In short, lockless techniques are quite useful and are heavily used. However, it is best if lockless techniques are hidden behind a well-defined API, such as the `inc_count()`, `memblock_alloc()`, `rcu_read_lock()`, and so on. The reason for this is that undisciplined use of lockless techniques is a good way to create difficult bugs. If you believe that finding and fixing such bugs is easier than avoiding them, please re-read Chapters 11 and 12.

14.2 Non-Blocking Synchronization

Never worry about theory as long as the machinery does what it's supposed to do.

Robert A. Heinlein

The term *non-blocking synchronization* (NBS) [Her90] describes seven classes of linearizable algorithms with

www.dbooks.org

differing *forward-progress guarantees* [ACHS13], which are as follows:

1. *Bounded wait-free synchronization*: Every thread will make progress within a specific finite period of time [Her91]. This level is widely considered to be unachievable, which might be why Alitarh et al. omitted it [ACHS13].
2. *Wait-free synchronization*: Every thread will make progress in finite time [Her93].
3. *Lock-free synchronization*: At least one thread will make progress in finite time [Her93].
4. *Obstruction-free synchronization*: Every thread will make progress in finite time in the absence of contention [HLM03].
5. *Clash-free synchronization*: At least one thread will make progress in finite time in the absence of contention [ACHS13].
6. *Starvation-free synchronization*: Every thread will make progress in finite time in the absence of failures [ACHS13].
7. *Deadlock-free synchronization*: At least one thread will make progress in finite time in the absence of failures [ACHS13].

NBS classes 1, 2 and 3 were first formulated in the early 1990s, class 4 was first formulated in the early 2000s, and class 5 was first formulated in 2013. The final two classes have seen informal use for a great many decades, but were reformulated in 2013.

In theory, any parallel algorithm can be cast into wait-free form, but there are a relatively small subset of NBS algorithms that are in common use. A few of these are listed in the following section.

14.2.1 Simple NBS

Perhaps the simplest NBS algorithm is atomic update of an integer counter using fetch-and-add (`atomic_add_return()`) primitives.

14.2.1.1 NBS Sets

Another simple NBS algorithm implements a set of integers in an array. Here the array index indicates a value that might be a member of the set and the array element indicates whether or not that value actually is a set member. The linearizability criterion for NBS algorithms requires that reads from and updates to the array either use atomic instructions or be accompanied by memory barriers, but in the not-uncommon case where linearizability is not important, simple volatile loads and stores suffice, for example, using `READ_ONCE()` and `WRITE_ONCE()`.

An NBS set may also be implemented using a bitmap, where each value that might be a member of the set corresponds to one bit. Reads and updates must normally be carried out via atomic bit-manipulation instructions, although compare-and-swap (`cmpxchg()` or CAS) instructions can also be used.

14.2.1.2 NBS Counters

The statistical counters algorithm discussed in Section 5.2 can be considered to be bounded-wait-free, but only by using a cute definitional trick in which the sum is considered to be approximate rather than exact.[1] Given sufficiently wide error bounds that are a function of the length of time that the `read_count()` function takes to sum the counters, it is not possible to prove that any non-linearizable behavior occurred. This definitely (if a bit artificially) classifies the statistical-counters algorithm as bounded wait-free. This algorithm is probably the most heavily used NBS algorithm in the Linux kernel.

14.2.1.3 Half-NBS Queue

Another common NBS algorithm is the atomic queue where elements are enqueued using an atomic exchange instruction [MS98b], followed by a store into the `->next` pointer of the new element's predecessor, as shown in Listing 14.1, which shows the userspace-RCU library implementation [Des09b]. Line 9 updates the tail pointer to reference the new element while returning a reference to its predecessor, which is stored in local variable `old_tail`. Line 10 then updates the predecessor's `->next` pointer to reference the newly added element, and finally line 11 returns an indication as to whether or not the queue was initially empty.

Although mutual exclusion is required to dequeue a single element (so that dequeue is blocking), it is possible to carry out a non-blocking removal of the entire contents of the queue. What is not possible is to dequeue any given element in a non-blocking manner: The enqueuer might have failed between lines 9 and 10 of the listing, so that the element in question is only partially enqueued.

[1] Citation needed. I heard of this trick verbally from Mark Moir.

Listing 14.1: NBS Enqueue Algorithm

```
1 static inline bool
2 ___cds_wfcq_append(struct cds_wfcq_head *head,
3                    struct cds_wfcq_tail *tail,
4                    struct cds_wfcq_node *new_head,
5                    struct cds_wfcq_node *new_tail)
6 {
7   struct cds_wfcq_node *old_tail;
8
9   old_tail = uatomic_xchg(&tail->p, new_tail);
10  CMM_STORE_SHARED(old_tail->next, new_head);
11  return old_tail != &head->node;
12 }
13
14 static inline bool
15 _cds_wfcq_enqueue(struct cds_wfcq_head *head,
16                   struct cds_wfcq_tail *tail,
17                   struct cds_wfcq_node *new_tail)
18 {
19   return ___cds_wfcq_append(head, tail,
20                             new_tail, new_tail);
21 }
```

This results in a half-NBS algorithm where enqueues are NBS but dequeues are blocking. This algorithm is nevertheless heavily used in practice, in part because most production software is not required to tolerate arbitrary fail-stop errors.

14.2.1.4 NBS Stack

Listing 14.2 shows the LIFO push algorithm, which boasts lock-free push and bounded wait-free pop (`lifo-push.c`), forming an NBS stack. The origins of this algorithm are unknown, but it was referred to in a patent granted in 1975 [BS75]. This patent was filed in 1973, a few months before your editor saw his first computer, which had but one CPU.

Lines 1–4 shows the `node_t` structure, which contains an arbitrary value and a pointer to the next structure on the stack and Line 7 shows the top-of-stack pointer.

The `list_push()` function spans lines 9–20. Line 11 allocates a new node and line 14 initializes it. Line 17 initializes the newly allocated node's `->next` pointer, and line 18 attempts to push it on the stack. If line 19 detects `cmpxchg()` failure, another pass through the loop retries. Otherwise, the new node has been successfully pushed, and this function returns to its caller. Note that line 19 resolves races in which two concurrent instances of `list_push()` attempt to push onto the stack. The `cmpxchg()` will succeed for one and fail for the other, causing the other to retry, thereby selecting an arbitrary order for the two node on the stack.

The `list_pop_all()` function spans lines 23–34. The `xchg()` statement on line 25 atomically removes all nodes on the stack, placing the head of the resulting list in local

Listing 14.2: NBS Stack Algorithm

```
1 struct node_t {
2   value_t val;
3   struct node_t *next;
4 };
5
6 // LIFO list structure
7 struct node_t* top;
8
9 void list_push(value_t v)
10 {
11   struct node_t *newnode = malloc(sizeof(*newnode));
12   struct node_t *oldtop;
13
14   newnode->val = v;
15   oldtop = READ_ONCE(top);
16   do {
17     newnode->next = oldtop;
18     oldtop = cmpxchg(&top, newnode->next, newnode);
19   } while (newnode->next != oldtop);
20 }
21
22
23 void list_pop_all(void (foo)(struct node_t *p))
24 {
25   struct node_t *p = xchg(&top, NULL);
26
27   while (p) {
28     struct node_t *next = p->next;
29
30     foo(p);
31     free(p);
32     p = next;
33   }
34 }
```

variable `p` and setting `top` to `NULL`. This atomic operation serializes concurrent calls to `list_pop_all()`: One of them will get the list, and the other a `NULL` pointer, at least assuming that there were no concurrent calls to `list_push()`.

An instance of `list_pop_all()` that obtains a non-empty list in `p` processes this list in the loop spanning lines 27–33. Line 28 prefetches the `->next` pointer, line 30 invokes the function referenced by `foo()` on the current node, line 31 frees the current node, and line 32 sets up `p` for the next pass through the loop.

But suppose that a pair of `list_push()` instances run concurrently with a `list_pop_all()` with a list initially containing a single node A. Here is one way that this scenario might play out:

1. The first `list_push()` instance pushes a new node B, executing through line 17, having just stored a pointer to node A into node B's `->next` pointer.

2. The `list_pop_all()` instance runs to completion, setting `top` to `NULL` and freeing node A.

www.dbooks.org

3. The second `list_push()` instance runs to completion, pushing a new node C, but happens to allocate the memory that used to belong to node A.

4. The first `list_push()` instance executes the `cmpxchg()` on line 18. Because new node C has the same address as the newly freed node A, this `cmpxchg()` succeeds and this `list_push()` instance runs to completion.

Note that both pushes and the popall all ran successfully despite the reuse of node A's memory. This is an unusual property: Most data structures require protection against what is often called the ABA problem.

But this property holds only for algorithm written in assembly language. The sad fact is that most languages (including C and C++) do not support pointers to lifetime-ended objects, such as the pointer to the old node A contained in node B's `->next` pointer. In fact, compilers are within their rights to assume that if two pointers (call them `p` and `q`) were returned from two different calls to `malloc()`, then those pointers must not be equal. Real compilers really will generate the constant `false` in response to a `p==q` comparison. A pointer to an object that has been freed, but whose memory has been reallocated for a compatibly typed object is termed a *zombie pointer*.

Many concurrent applications avoid this problem by carefully hiding the memory allocator from the compiler, thus preventing the compiler from making inappropriate assumptions. This obfuscatory approach currently works in practice, but might well one day fall victim to increasingly aggressive optimizers. There is work underway in both the C and C++ standards committees to address this problem [MMS19, MMM+20]. In the meantime, please exercise great care when coding ABA-tolerant algorithms.

Quick Quiz 14.1: So why not ditch antique languages like C and C++ for something more modern? ■

14.2.2 Applicability of NBS Benefits

The most heavily cited NBS benefits stem from its forward-progress guarantees, its tolerance of fail-stop bugs, and from its linearizability. Each of these is discussed in one of the following sections.

14.2.2.1 NBS Forward Progress Guarantees

NBS's forward-progress guarantees have caused many to suggest its use in real-time systems, and NBS algorithms are in fact used in a great many such systems. However, it is important to note that forward-progress guarantees are largely orthogonal to those that form the basis of real-time programming:

1. Real-time forward-progress guarantees usually have some definite time associated with them, for example, "scheduling latency must be less than 100 microseconds." In contrast, the most popular forms of NBS only guarantees that progress will be made in finite time, with no definite bound.

2. Real-time forward-progress guarantees are often probabilistic, as in the soft-real-time guarantee that "at least 99.9 % of the time, scheduling latency must be less than 100 microseconds." In contrast, many of NBS's forward-progress guarantees are unconditional.

3. Real-time forward-progress guarantees are often conditioned on environmental constraints, for example, only being honored: (1) For the highest-priority tasks, (2) When each CPU spends at least a certain fraction of its time idle, and (3) When I/O rates are below some specified maximum. In contrast, NBS's forward-progress guarantees are often unconditional, although recent NBS work accommodates conditional guarantees [ACHS13].

4. An important component of a real-time program's environment is the scheduler. NBS algorithms assume a worst-case *demonic scheduler*. In contrast, real-time systems assume that the scheduler is doing its level best to satisfy any scheduling constraints it knows about, and, in the absence of such constraints, its level best to honor process priorities and to provide fair scheduling to processes of the same priority. This assumption of a non-demonic scheduler allows real-time programs to use simpler algorithms than those required for NBS [ACHS13, Bra11].

5. Real-time forward-progress guarantees usually apply only in the absence of software bugs. In contrast, many classes of NBS guarantees apply even in the face of fail-stop bugs.

6. NBS forward-progress guarantee classes imply linearizability. In contrast, real-time forward progress guarantees are often independent of ordering constraints such as linearizability.

To reiterate, despite these differences, a number of NBS algorithms are extremely useful in real-time programs.

14.2.2.2 NBS Fail-Stop Tolerance

Of the classes of NBS algorithms, wait-free synchronization (bounded or otherwise), lock-free synchronization, obstruction-free synchronization, and clash-free synchronization guarantee forward progress even in the presence of fail-stop bugs. An example fail-stop bug might cause some thread to be preempted indefinitely. As we will see, this fail-stop-tolerant property can be useful, but the fact is that composing a set of fail-stop-tolerant mechanisms does not necessarily result in a fail-stop-tolerant system. To see this, consider a system made up of a series of wait-free queues, where an element is removed from one queue in the series, processed, and then added to the next queue.

If a thread is preempted in the midst of a queuing operation, in theory all is well because the wait-free nature of the queue will guarantee forward progress. But in practice, the element being processed is lost because the fail-stop-tolerant nature of the wait-free queues does not extend to the code using those queues.

Nevertheless, there are a few applications where NBS's rather limited fail-stop-tolerance is useful. For example, in some network-based or web applications, a fail-stop event will eventually result in a retransmission, which will restart any work that was lost due to the fail-stop event. Systems running such applications can therefore be heavily loaded, even to the point where the scheduler can no longer provide any reasonable fairness guarantee. In constrast, if a thread fail-stops while holding a lock, the application might need to be restarted. Nevertheless, NBS is not a panacea even within this restricted area, due to the possibility of spurious retransmissions due to pure scheduling delays. In some cases, it may be more efficient to reduce the load to avoid queueing delays, which will also improve the scheduler's ability to provide fair access, reducing or even eliminating the fail-stop events, thus reducing the number of retry operations, in turn further reducing the load.

14.2.2.3 NBS Linearizability

It is important to note that linearizability can be quite useful, especially when analyzing concurrent code made up of strict locking and fully ordered atomic operations.[2] Furthermore, this handling of fully ordered atomic operations automatically covers simple NBS algorithms.

However, the linearization points of a complex NBS algorithms are often buried deep within that algorithm, and thus not visible to users of a library function implementing a part of such an algorithm. Therefore, any claims that users benefit from the linearizability properties of complex NBS algorithms should be regarded with deep suspicion [HKLP12].

It is sometimes asserted that linearizability is necessary for developers to produce proofs of correctness for their concurrent code. However, such proofs are the exception rather than the rule, and modern developers who do produce proofs often use modern proof techniques that do not depend on linearizability. Furthermore, developers frequently use modern proof techniques that do not require a full specification, given that developers often learn their specification after the fact, one bug at a time. A few such proof techniques were discussed in Chapter 12.[3]

It is often asserted that linearizability maps well to sequential specifications, which are said to be more natural than are concurrent specifications [RR20]. But this assertion fails to account for our highly concurrent objective universe. This universe can only be expected to select for ability to cope with concurrency, especially for those participating in team sports or overseeing small children. In addition, given that the teaching of sequential computing is still believed to be somewhat of a black art [PBCE20], it is reasonable to expect that teaching of concurrent computing is in a similar state of disarray. Therefore, focusing on only one proof technique is unlikely to be a good way forward.

Again, please understand that linearizability is quite useful in many situations. Then again, so is that venerable tool, the hammer. But there comes a point in the field of computing where one should put down the hammer and pick up a keyboard. Similarly, it appears that there are times when linearizability is not the best tool for the job.

To their credit, there are some linearizability advocates who are aware of some of its shortcomings [RR20]. There are also proposals to extend linearizability, for example, interval-linearizability, which is intended to handle the common case of operations that require non-zero time to complete [CnRR18]. It remains to be seen whether these proposals will result in theories able to handle modern concurrent software artifacts, especially given that several of the proof techniques discussed in Chapter 12 already handle many modern concurrent software artifacts.

[2] For example, the Linux kernel's value-returning atomic operations.

[3] A memorable verbal discussion with an advocate of linearizability resulted in question: "So the reason linearizability is important is to rescue 1980s proof techniques?" The advocate immediately replied in the affirmative, then spent some time disparaging a particular modern proof technique. Oddly enough, that technique was one of those successfully applied to Linux-kernel RCU.

www.dbooks.org

14.2.3 NBS Discussion

It is possible to create fully non-blocking queues [MS96], however, such queues are much more complex than the half-NBS algorithm outlined above. The lesson here is to carefully consider your actual requirements. Relaxing irrelevant requirements can often result in great improvements in simplicity, performance, and scalability.

Recent research points to another important way to relax requirements. It turns out that systems providing fair scheduling can enjoy most of the benefits of wait-free synchronization even when running algorithms that provide only non-blocking synchronization, both in theory [ACHS13] and in practice [AB13]. Because most schedulers used in production do in fact provide fairness, the more-complex algorithms providing wait-free synchronization usually provide no practical advantages over simpler and faster non-wait-free algorithms.

Interestingly enough, fair scheduling is but one beneficial constraint that is often respected in practice. Other sets of constraints can permit blocking algorithms to achieve deterministic real-time response. For example, given: (1) Fair locks granted in FIFO order within a given priority level, (2) Priority inversion avoidance (for example, priority inheritance [TS95, WTS96] or priority ceiling), (3) A bounded number of threads, (4) Bounded critical section durations, (5) Bounded load, and (6) Absence of fail-stop bugs, lock-based applications can provide deterministic response times [Bra11, SM04a]. This approach of course blurs the distinction between blocking and wait-free synchronization, which is all to the good. Hopefully theoretical frameworks will continue to improve their ability to describe software actually used in practice.

Those who feel that theory should lead the way are referred to the inimitable Peter Denning, who said of operating systems: "Theory follows practice" [Den15], or to the eminent Tony Hoare, who said of the whole of engineering: "In all branches of engineering science, the engineering starts before the science; indeed, without the early products of engineering, there would be nothing for the scientist to study!" [Mor07]. However, once an appropriate body of theory becomes available,[4] it is wise to make use of it.

[4] Note well that the first *appropriate* body of theory is often one thing and the first *proposed* body of theory quite another.

14.3 Parallel Real-Time Computing

> One always has time enough if one applies it well.
>
> *Johann Wolfgang von Göthe*

An important emerging area in computing is that of parallel real-time computing. Section 14.3.1 looks at a number of definitions of "real-time computing", moving beyond the usual sound bites to more meaningful criteria. Section 14.3.2 surveys the sorts of applications that need real-time response. Section 14.3.3 notes that parallel real-time computing is upon us, and discusses when and why parallel real-time computing can be useful. Section 14.3.4 gives a brief overview of how parallel real-time systems may be implemented, with Sections 14.3.5 and 14.3.6 focusing on operating systems and applications, respectively. Finally, Section 14.3.7 outlines how to decide whether or not your application needs real-time facilities.

14.3.1 What is Real-Time Computing?

One traditional way of classifying real-time computing is into the categories of *hard real time* and *soft real time*, where the macho hard real-time applications never miss their deadlines, but the wimpy soft real-time applications miss their deadlines quite often.

14.3.1.1 Soft Real Time

It should be easy to see problems with this definition of soft real time. For one thing, by this definition, *any* piece of software could be said to be a soft real-time application: "My application computes million-point Fourier transforms in half a picosecond." "No way!!! The clock cycle on this system is more than *three hundred* picoseconds!" "Ah, but it is a *soft* real-time application!" If the term "soft real time" is to be of any use whatsoever, some limits are clearly required.

We might therefore say that a given soft real-time application must meet its response-time requirements at least some fraction of the time, for example, we might say that it must execute in less than 20 microseconds 99.9 % of the time.

This of course raises the question of what is to be done when the application fails to meet its response-time requirements. The answer varies with the application, but one possibility is that the system being controlled has sufficient stability and inertia to render harmless the occasional late control action. Another possibility is that



Figure 14.1: Real-Time Response, Meet Hammer

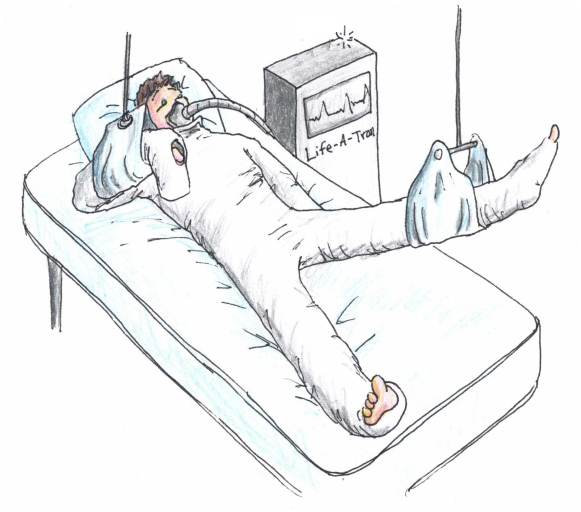


Figure 14.2: Real-Time Response: Hardware Matters

the application has two ways of computing the result, a fast and deterministic but inaccurate method on the one hand and a very accurate method with unpredictable compute time on the other. One reasonable approach would be to start both methods in parallel, and if the accurate method fails to finish in time, kill it and use the answer from the fast but inaccurate method. One candidate for the fast but inaccurate method is to take no control action during the current time period, and another candidate is to take the same control action as was taken during the preceding time period.

In short, it does not make sense to talk about soft real time without some measure of exactly how soft it is.

14.3.1.2 Hard Real Time

In contrast, the definition of hard real time is quite definite. After all, a given system either always meets its deadlines or it doesn't. Unfortunately, a strict application of this definition would mean that there can never be any hard real-time systems. The reason for this is fancifully depicted in Figure 14.1. Yes, you could construct a more robust system, perhaps with redundancy. But your adversary can always get a bigger hammer.

Then again, perhaps it is unfair to blame the software for what is clearly not just a hardware problem, but a bona fide big-iron hardware problem at that.[5] This suggests that we define hard real-time software as software that will always meet its deadlines, but only in the absence of a hardware failure. Unfortunately, failure is not always an option, as fancifully depicted in Figure 14.2. We simply cannot expect the poor gentleman depicted in that figure to be reassured our saying "Rest assured that if a missed deadline results in your tragic death, it most certainly will not have been due to a software problem!" Hard real-time response is a property of the entire system, not just of the software.

But if we cannot demand perfection, perhaps we can make do with notification, similar to the soft real-time approach noted earlier. Then if the Life-a-Tron in Figure 14.2 is about to miss its deadline, it can alert the hospital staff.

Unfortunately, this approach has the trivial solution fancifully depicted in Figure 14.3. A system that always immediately issues a notification that it won't be able to meet its deadline complies with the letter of the law, but is completely useless. There clearly must also be a requirement that the system meets its deadline some fraction of the time, or perhaps that it be prohibited from missing its deadlines on more than a certain number of consecutive operations.

We clearly cannot take a sound-bite approach to either hard or soft real time. The next section therefore takes a more real-world approach.

14.3.1.3 Real-World Real Time

Although sentences like "Hard real-time systems *always* meet their deadlines!" are catchy and easy to memorize, something else is needed for real-world real-time systems. Although the resulting specifications are harder to memo-

[5] Or, given modern hammers, a big-steel problem.

www.dbooks.org



Figure 14.3: Real-Time Response: Notification Insufficient

rize, they can simplify construction of a real-time system by imposing constraints on the environment, the workload, and the real-time application itself.

Environmental Constraints Constraints on the environment address the objection to open-ended promises of response times implied by "hard real time". These constraints might specify permissible operating temperatures, air quality, levels and types of electromagnetic radiation, and, to Figure 14.1's point, levels of shock and vibration.

Of course, some constraints are easier to meet than others. Any number of people have learned the hard way that commodity computer components often refuse to operate at sub-freezing temperatures, which suggests a set of climate-control requirements.

An old college friend once had the challenge of operating a real-time system in an atmosphere featuring some rather aggressive chlorine compounds, a challenge that he wisely handed off to his colleagues designing the hardware. In effect, my colleague imposed an atmospheric-composition constraint on the environment immediately surrounding the computer, a constraint that the hardware designers met through use of physical seals.

Another old college friend worked on a computer-controlled system that sputtered ingots of titanium using an industrial-strength arc in a vacuum. From time to time, the arc would decide that it was bored with its path through the ingot of titanium and choose a far shorter and more entertaining path to ground. As we all learned in our physics classes, a sudden shift in the flow of electrons creates an electromagnetic wave, with larger shifts in larger flows creating higher-power electromagnetic waves. And in this case, the resulting electromagnetic pulses were sufficient to induce a quarter of a volt potential difference in the leads of a small "rubber ducky" antenna located more than 400 meters away. This meant that nearby conductors experienced higher voltages, courtesy of the inverse-square law. This included those conductors making up the computer controlling the sputtering process. In particular, the voltage induced on that computer's reset line was sufficient to actually reset the computer, mystifying everyone involved. This situation was addressed using hardware, including some elaborate shielding and a fiber-optic network with the lowest bitrate I have ever heard of, namely 9600 baud. Less spectacular electromagnetic environments can often be handled by software through use of error detection and correction codes. That said, it is important to remember that although error detection and correction codes can reduce failure rates, they normally cannot reduce them all the way down to zero, which can present yet another obstacle to achieving hard real-time response.

There are also situations where a minimum level of energy is required, for example, through the power leads of the system and through the devices through which the system is to communicate with that portion of the outside world that is to be monitored or controlled.

Quick Quiz 14.2: But what about battery-powered systems? They don't require energy flowing into the system as a whole. ■

A number of systems are intended to operate in environments with impressive levels of shock and vibration, for example, engine control systems. More strenuous requirements may be found when we move away from continuous vibrations to intermittent shocks. For example, during my undergraduate studies, I encountered an old Athena ballistics computer, which was designed to continue operating normally even if a hand grenade went off nearby.[6] And finally, the "black boxes" used in airliners must continue operating before, during, and after a crash.

Of course, it is possible to make hardware more robust against environmental shocks and insults. Any number of ingenious mechanical shock-absorbing devices can reduce the effects of shock and vibration, multiple layers of shielding can reduce the effects of low-energy electromagnetic radiation, error-correction coding can reduce the effects of high-energy radiation, various potting and sealing techniques can reduce the effect of air quality, and any number of heating and cooling systems can counter the effects of temperature. In extreme cases, triple modular redundancy can reduce the probability that a fault in one part of the system will result in incorrect behavior from the overall system. However, all of these methods have one thing in common: Although they can reduce the probability of failure, they cannot reduce it to zero.

These environmental challenges are often met via robust hardware, however, the workload and application constraints in the next two sections are often handled in software.

Workload Constraints Just as with people, it is often possible to prevent a real-time system from meeting its deadlines by overloading it. For example, if the system is being interrupted too frequently, it might not have sufficient CPU bandwidth to handle its real-time application. A hardware solution to this problem might limit the rate at which interrupts were delivered to the system. Possible software solutions include disabling interrupts for some time if they are being received too frequently, resetting the device generating too-frequent interrupts, or even avoiding interrupts altogether in favor of polling.

Overloading can also degrade response times due to queueing effects, so it is not unusual for real-time systems to overprovision CPU bandwidth, so that a running system has (say) 80 % idle time. This approach also applies to storage and networking devices. In some cases, separate storage and networking hardware might be reserved for the sole use of high-priority portions of the real-time application. In short, it is not unusual for this hardware to be mostly idle, given that response time is more important than throughput in real-time systems.

Quick Quiz 14.3: But given the results from queueing theory, won't low utilization merely improve the average response time rather than improving the worst-case response time? And isn't worst-case response time all that most real-time systems really care about? ■

Of course, maintaining sufficiently low utilization requires great discipline throughout the design and implementation. There is nothing quite like a little feature creep to destroy deadlines.

Application Constraints It is easier to provide bounded response time for some operations than for others. For example, it is quite common to see response-time specifications for interrupts and for wake-up operations, but quite rare for (say) filesystem unmount operations. One reason for this is that it is quite difficult to bound the amount of work that a filesystem-unmount operation might need to do, given that the unmount is required to flush all of that filesystem's in-memory data to mass storage.

This means that real-time applications must be confined to operations for which bounded latencies can reasonably be provided. Other operations must either be pushed out into the non-real-time portions of the application or forgone entirely.

There might also be constraints on the non-real-time portions of the application. For example, is the non-real-time application permitted to use the CPUs intended for the real-time portion? Are there time periods during which the real-time portion of the application is expected to be unusually busy, and if so, is the non-real-time portion of the application permitted to run at all during those times? Finally, by what amount is the real-time portion of the application permitted to degrade the throughput of the non-real-time portion?

Real-World Real-Time Specifications As can be seen from the preceding sections, a real-world real-time specification needs to include constraints on the environment, on the workload, and on the application itself. In addition, for the operations that the real-time portion of the application

[6] Decades later, the acceptance tests for some types of computer systems involve large detonations, and some types of communications networks must deal with what is delicately termed "ballistic jamming."

www.dbooks.org

is permitted to make use of, there must be constraints on the hardware and software implementing those operations.

For each such operation, these constraints might include a maximum response time (and possibly also a minimum response time) and a probability of meeting that response time. A probability of 100 % indicates that the corresponding operation must provide hard real-time service.

In some cases, both the response times and the required probabilities of meeting them might vary depending on the parameters to the operation in question. For example, a network operation over a local LAN would be much more likely to complete in (say) 100 microseconds than would that same network operation over a transcontinental WAN. Furthermore, a network operation over a copper or fiber LAN might have an extremely high probability of completing without time-consuming retransmissions, while that same networking operation over a lossy WiFi network might have a much higher probability of missing tight deadlines. Similarly, a read from a tightly coupled solid-state disk (SSD) could be expected to complete much more quickly than that same read to an old-style USB-connected rotating-rust disk drive.[7]

Some real-time applications pass through different phases of operation. For example, a real-time system controlling a plywood lathe that peels a thin sheet of wood (called "veneer") from a spinning log must: (1) Load the log into the lathe, (2) Position the log on the lathe's chucks so as to expose the largest cylinder contained within that log to the blade, (3) Start spinning the log, (4) Continuously vary the knife's position so as to peel the log into veneer, (5) Remove the remaining core of the log that is too small to peel, and (6) Wait for the next log. Each of these six phases of operation might well have its own set of deadlines and environmental constraints, for example, one would expect phase 4's deadlines to be much more severe than those of phase 6, as in milliseconds rather than seconds. One might therefore expect that low-priority work would be performed in phase 6 rather than in phase 4. In any case, careful choices of hardware, drivers, and software configuration would be required to support phase 4's more severe requirements.

A key advantage of this phase-by-phase approach is that the latency budgets can be broken down, so that the application's various components can be developed independently, each with its own latency budget. Of course, as with any other kind of budget, there will likely be the occasional conflict as to which component gets which fraction of the overall budget, and as with any other kind of budget, strong leadership and a sense of shared goals can help to resolve these conflicts in a timely fashion. And, again as with other kinds of technical budget, a strong validation effort is required in order to ensure proper focus on latencies and to give early warning of latency problems. A successful validation effort will almost always include a good test suite, which might be unsatisfying to the theorists, but has the virtue of helping to get the job done. As a point of fact, as of early 2021, most real-world real-time system use an acceptance test rather than formal proofs.

However, the widespread use of test suites to validate real-time systems does have a very real disadvantage, namely that real-time software is validated only on specific configurations of hardware and software. Adding additional configurations requires additional costly and time-consuming testing. Perhaps the field of formal verification will advance sufficiently to change this situation, but as of early 2021, rather large advances are required.

Quick Quiz 14.4: Formal verification is already quite capable, benefiting from decades of intensive study. Are additional advances *really* required, or is this just a practitioner's excuse to continue to lazily ignore the awesome power of formal verification? ■

In addition to latency requirements for the real-time portions of the application, there will likely be performance and scalability requirements for the non-real-time portions of the application. These additional requirements reflect the fact that ultimate real-time latencies are often attained by degrading scalability and average performance.

Software-engineering requirements can also be important, especially for large applications that must be developed and maintained by large teams. These requirements often favor increased modularity and fault isolation.

This is a mere outline of the work that would be required to specify deadlines and environmental constraints for a production real-time system. It is hoped that this outline clearly demonstrates the inadequacy of the sound-bite-based approach to real-time computing.

14.3.2 Who Needs Real-Time?

It is possible to argue that all computing is in fact real-time computing. For one example, when you purchase a birthday gift online, you expect the gift to arrive before the recipient's birthday. And in fact even turn-of-the-

[7] Important safety tip: Worst-case response times from USB devices can be extremely long. Real-time systems should therefore take care to place any USB devices well away from critical paths.

millennium web services observed sub-second response constraints [Boh01], and requirements have not eased with the passage of time [DHJ+07]. It is nevertheless useful to focus on those real-time applications whose response-time requirements cannot be achieved straightforwardly by non-real-time systems and applications. Of course, as hardware costs decrease and bandwidths and memory sizes increase, the line between real-time and non-real-time will continue to shift, but such progress is by no means a bad thing.

Quick Quiz 14.5: Differentiating real-time from non-real-time based on what can "be achieved straightforwardly by non-real-time systems and applications" is a travesty! There is absolutely no theoretical basis for such a distinction!!! Can't we do better than that??? ■

Real-time computing is used in industrial-control applications, ranging from manufacturing to avionics; scientific applications, perhaps most spectacularly in the adaptive optics used by large Earth-bound telescopes to de-twinkle starlight; military applications, including the afore-mentioned avionics; and financial-services applications, where the first computer to recognize an opportunity is likely to reap most of the profit. These four areas could be characterized as "in search of production", "in search of life", "in search of death", and "in search of money".

Financial-services applications differ subtlely from applications in the other three categories in that money is non-material, meaning that non-computational latencies are quite small. In contrast, mechanical delays inherent in the other three categories provide a very real point of diminishing returns beyond which further reductions in the application's real-time response provide little or no benefit. This means that financial-services applications, along with other real-time information-processing applications, face an arms race, where the application with the lowest latencies normally wins. Although the resulting latency requirements can still be specified as described in Paragraph "Real-World Real-Time Specifications" on Page 279, the unusual nature of these requirements has led some to refer to financial and information-processing applications as "low latency" rather than "real time".

Regardless of exactly what we choose to call it, there is substantial need for real-time computing [Pet06, Inm07].

14.3.3 Who Needs Parallel Real-Time?

It is less clear who really needs parallel real-time computing, but the advent of low-cost multicore systems has brought it to the fore regardless. Unfortunately, the traditional mathematical basis for real-time computing assumes single-CPU systems, with a few exceptions that prove the rule [Bra11]. Fortunately, there are a couple of ways of squaring modern computing hardware to fit the real-time mathematical circle, and a few Linux-kernel hackers have been encouraging academics to make this transition [dOCdO19, Gle10].

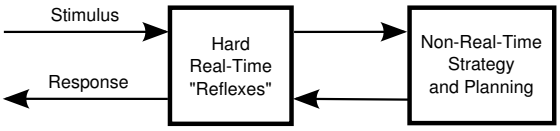


Figure 14.4: Real-Time Reflexes

One approach is to recognize the fact that many real-time systems resemble biological nervous systems, with responses ranging from real-time reflexes to non-real-time strategizing and planning, as depicted in Figure 14.4. The hard real-time reflexes, which read from sensors and control actuators, run real-time on a single CPU or on special-purpose hardware such as an FPGA. The non-real-time strategy and planning portion of the application runs on the remaining CPUs. Strategy and planning activities might include statistical analysis, periodic calibration, user interface, supply-chain activities, and preparation. For an example of high-compute-load preparation activities, think back to the veneer-peeling application discussed in Paragraph "Real-World Real-Time Specifications" on Page 279. While one CPU is attending to the high-speed real-time computations required to peel one log, the other CPUs might be analyzing the size and shape of the next log in order to determine how to position the next log so as to obtain the largest cylinder of high-quality wood. It turns out that many applications have non-real-time and real-time components [BMP08], so this approach can often be used to allow traditional real-time analysis to be combined with modern multicore hardware.

Another trivial approach is to shut off all but one hardware thread so as to return to the settled mathematics of uniprocessor real-time computing. However, this approach gives up potential cost and energy-efficiency advantages. That said, obtaining these advantages requires overcoming the parallel performance obstacles covered in Chapter 3, and not merely on average, but instead in the worst case.

Implementing parallel real-time systems can therefore be quite a challenge. Ways of meeting this challenge are outlined in the following section.

www.dbooks.org

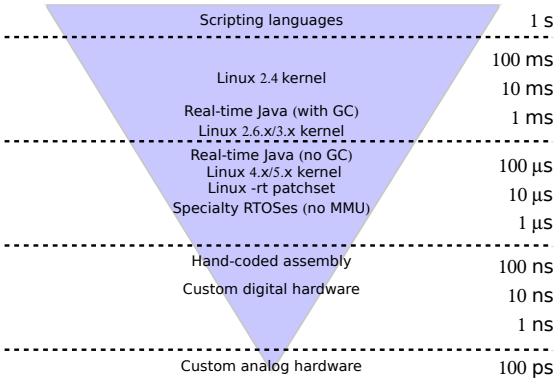


Figure 14.5: Real-Time Response Regimes

14.3.4 Implementing Parallel Real-Time Systems

We will look at two major styles of real-time systems, event-driven and polling. An event-driven real-time system remains idle much of the time, responding in real time to events passed up through the operating system to the application. Alternatively, the system could instead be running a background non-real-time workload. A polling real-time system features a real-time thread that is CPU bound, running in a tight loop that polls inputs and updates outputs on each pass. This tight polling loop often executes entirely in user mode, reading from and writing to hardware registers that have been mapped into the user-mode application's address space. Alternatively, some applications place the polling loop into the kernel, for example, using loadable kernel modules.

Regardless of the style chosen, the approach used to implement a real-time system will depend on the deadlines, for example, as shown in Figure 14.5. Starting from the top of this figure, if you can live with response times in excess of one second, you might well be able to use scripting languages to implement your real-time application—and scripting languages are in fact used surprisingly often, not that I necessarily recommend this practice. If the required latencies exceed several tens of milliseconds, old 2.4 versions of the Linux kernel can be used, not that I necessarily recommend this practice, either. Special real-time Java implementations can provide real-time response latencies of a few milliseconds, even when the garbage collector is used. The Linux 2.6.x and 3.x kernels can provide real-time latencies of a few hundred microseconds if carefully configured, tuned, and run on real-time friendly hardware. Special real-time Java implementations can provide real-time latencies below 100 microseconds if use of the garbage collector is carefully avoided. (But note that avoiding the garbage collector means also avoiding Java's large standard libraries, thus also avoiding Java's productivity advantages.) The Linux 4.x and 5.x kernels can provide deep sub-hundred-millisecond latencies, but with all the same caveats as for the 2.6.x and 3.x kernels. A Linux kernel incorporating the -rt patchset can provide latencies well below 20 microseconds, and specialty real-time operating systems (RTOSes) running without MMUs can provide sub-ten-microsecond latencies. Achieving sub-microsecond latencies typically requires hand-coded assembly or even special-purpose hardware.

Of course, careful configuration and tuning are required all the way down the stack. In particular, if the hardware or firmware fails to provide real-time latencies, there is nothing that the software can do to make up for the lost time. Worse yet, high-performance hardware sometimes sacrifices worst-case behavior to obtain greater throughput. In fact, timings from tight loops run with interrupts disabled can provide the basis for a high-quality random-number generator [MOZ09]. Furthermore, some firmware does cycle-stealing to carry out various housekeeping tasks, in some cases attempting to cover its tracks by reprogramming the victim CPU's hardware clocks. Of course, cycle stealing is expected behavior in virtualized environment, but people are nevertheless working towards real-time response in virtualized environments [Gle12, Kis14]. It is therefore critically important to evaluate your hardware's and firmware's real-time capabilities.

But given competent real-time hardware and firmware, the next layer up the stack is the operating system, which is covered in the next section.

14.3.5 Implementing Parallel Real-Time Operating Systems

There are a number of strategies that may be used to implement a real-time system. One approach is to port a general-purpose non-real-time OS on top of a special purpose real-time operating system (RTOS), as shown in Figure 14.6. The green "Linux Process" boxes represent non-real-time processes running on the Linux kernel, while the yellow "RTOS Process" boxes represent real-time processes running on the RTOS.

This was a very popular approach before the Linux kernel gained real-time capabilities, and is still in use [xen14, Yod04b]. However, this approach requires that the application be split into one portion that runs on

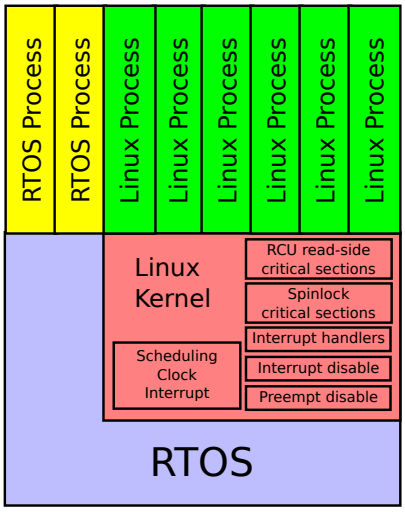


Figure 14.6: Linux Ported to RTOS

the RTOS and another that runs on Linux. Although it is possible to make the two environments look similar, for example, by forwarding POSIX system calls from the RTOS to a utility thread running on Linux, there are invariably rough edges.

In addition, the RTOS must interface to both the hardware and to the Linux kernel, thus requiring significant maintenance with changes in both hardware and kernel. Furthermore, each such RTOS often has its own system-call interface and set of system libraries, which can balkanize both ecosystems and developers. In fact, these problems seem to be what drove the combination of RTOSes with Linux, as this approach allowed access to the full real-time capabilities of the RTOS, while allowing the application's non-real-time code full access to Linux's open-source ecosystem.

Although pairing RTOSes with the Linux kernel was a clever and useful short-term response during the time that the Linux kernel had minimal real-time capabilities, it also motivated adding real-time capabilities to the Linux kernel. Progress towards this goal is shown in Figure 14.7. The upper row shows a diagram of the Linux kernel with preemption disabled, thus having essentially no real-time capabilities. The middle row shows a set of diagrams showing the increasing real-time capabilities of the mainline Linux kernel with preemption enabled. Finally, the bottom row shows a diagram of the Linux kernel with the -rt patchset applied, maximizing real-time capabilities. Functionality from the -rt patchset is added to mainline, hence the increasing capabilities of the mainline Linux kernel over time. Nevertheless, the most demanding real-time applications continue to use the -rt patchset.

The non-preemptible kernel shown at the top of Figure 14.7 is built with `CONFIG_PREEMPT=n`, so that execution within the Linux kernel cannot be preempted. This means that the kernel's real-time response latency is bounded below by the longest code path in the Linux kernel, which is indeed long. However, user-mode execution is preemptible, so that one of the real-time Linux processes shown in the upper right may preempt any of the non-real-time Linux processes shown in the upper left anytime the non-real-time process is executing in user mode.

The middle row of Figure 14.7 shows three stages (from left to right) in the development of Linux's preemptible kernels. In all three stages, most process-level code within the Linux kernel can be preempted. This of course greatly improves real-time response latency, but preemption is still disabled within RCU read-side critical sections, spinlock critical sections, interrupt handlers, interrupt-disabled code regions, and preempt-disabled code regions, as indicated by the red boxes in the left-most diagram in the middle row of the figure. The advent of preemptible RCU allowed RCU read-side critical sections to be preempted, as shown in the central diagram, and the advent of threaded interrupt handlers allowed device-interrupt handlers to be preempted, as shown in the right-most diagram. Of course, a great deal of other real-time functionality was added during this time, however, it cannot be as easily represented on this diagram. It will instead be discussed in Section 14.3.5.1.

The bottom row of Figure 14.7 shows the -rt patchset, which features threaded (and thus preemptible) interrupt handlers for many devices, which also allows the corresponding "interrupt-disabled" regions of these drivers to be preempted. These drivers instead use locking to coordinate the process-level portions of each driver with its threaded interrupt handlers. Finally, in some cases, disabling of preemption is replaced by disabling of migration. These measures result in excellent response times in many systems running the -rt patchset [RMF19, dOCdO19].

A final approach is simply to get everything out of the way of the real-time process, clearing all other processing off of any CPUs that this process needs, as shown in Figure 14.8. This was implemented in the 3.10 Linux kernel via the `CONFIG_NO_HZ_FULL` Kconfig parameter [Cor13, Wei12]. It is important to note that this approach requires at least one *housekeeping CPU* to do background processing, for example running kernel dae-

www.dbooks.org

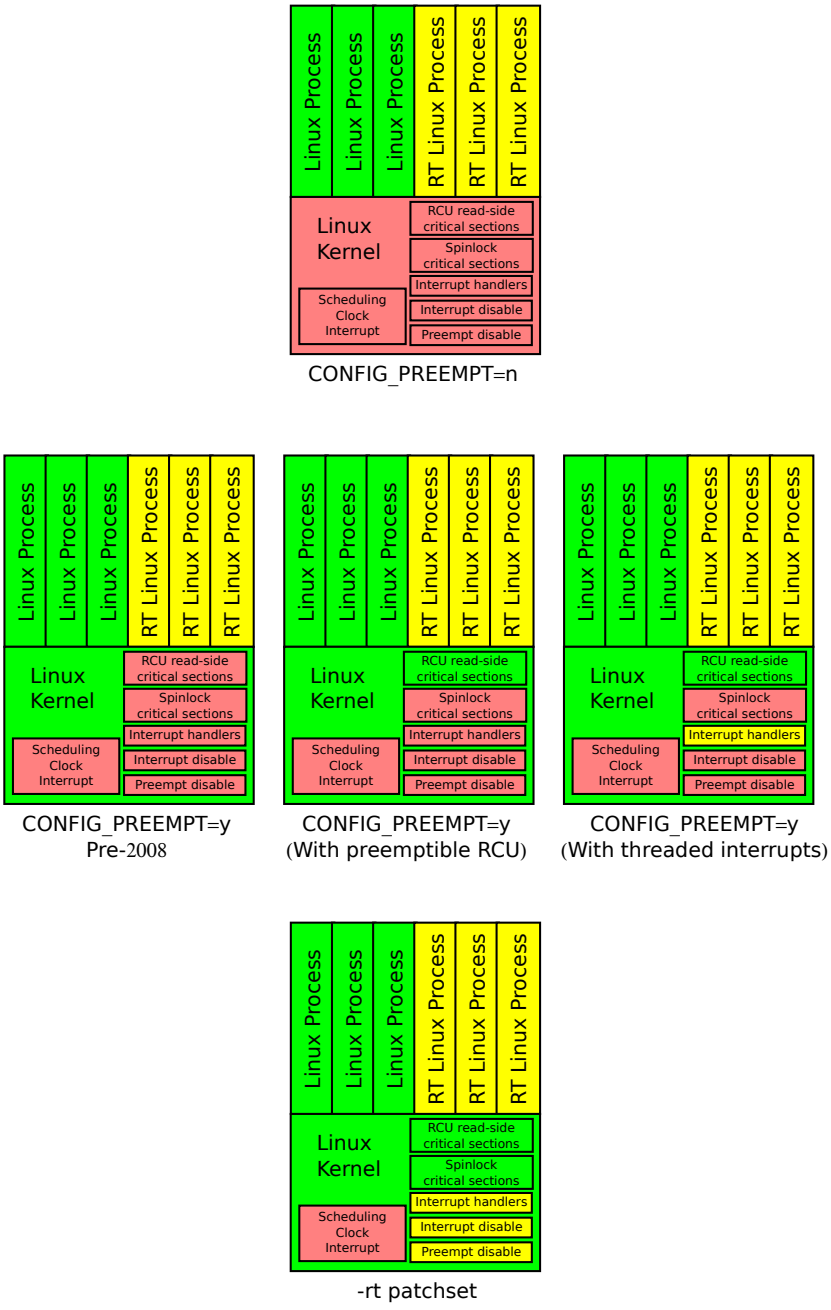


Figure 14.7: Linux-Kernel Real-Time Implementations

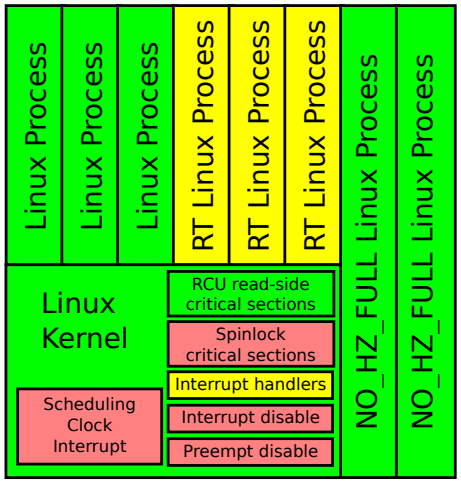


Figure 14.8: CPU Isolation

mons. However, when there is only one runnable task on a given non-housekeeping CPU, scheduling-clock interrupts are shut off on that CPU, removing an important source of interference and *OS jitter*. With a few exceptions, the kernel does not force other processing off of the non-housekeeping CPUs, but instead simply provides better performance when only one runnable task is present on a given CPU. Any number of userspace tools may be used to force a given CPU to have no more that one runnable task. If configured properly, a non-trivial undertaking, `CONFIG_NO_HZ_FULL` offers real-time threads levels of performance that come close to those of bare-metal systems [ACA+18].

There has of course been much debate over which of these approaches is best for real-time systems, and this debate has been going on for quite some time [Cor04a, Cor04c]. As usual, the answer seems to be "It depends," as discussed in the following sections. Section 14.3.5.1 considers event-driven real-time systems, and Section 14.3.5.2 considers real-time systems that use a CPU-bound polling loop.

14.3.5.1 Event-Driven Real-Time Support

The operating-system support required for event-driven real-time applications is quite extensive, however, this section will focus on only a few items, namely timers, threaded interrupts, priority inheritance, preemptible RCU, and preemptible spinlocks.

Timers are clearly critically important for real-time operations. After all, if you cannot specify that something be done at a specific time, how are you going to respond by that time? Even in non-real-time systems, large numbers of timers are generated, so they must be handled extremely efficiently. Example uses include retransmit timers for TCP connections (which are almost always canceled before they have a chance to fire),[8] timed delays (as in `sleep(1)`, which are rarely canceled), and timeouts for the `poll()` system call (which are often canceled before they have a chance to fire). A good data structure for such timers would therefore be a priority queue whose addition and deletion primitives were fast and $O(1)$ in the number of timers posted.

The classic data structure for this purpose is the *calendar queue*, which in the Linux kernel is called the `timer wheel`. This age-old data structure is also heavily used in discrete-event simulation. The idea is that time is quantized, for example, in the Linux kernel, the duration of the time quantum is the period of the scheduling-clock interrupt. A given time can be represented by an integer, and any attempt to post a timer at some non-integral time will be rounded to a convenient nearby integral time quantum.

One straightforward implementation would be to allocate a single array, indexed by the low-order bits of the time. This works in theory, but in practice systems create large numbers of long-duration timeouts (for example, the two-hour keepalive timeouts for TCP sessions) that are almost always canceled. These long-duration timeouts cause problems for small arrays because much time is wasted skipping timeouts that have not yet expired. On the other hand, an array that is large enough to gracefully accommodate a large number of long-duration timeouts would consume too much memory, especially given that performance and scalability concerns require one such array for each and every CPU.

A common approach for resolving this conflict is to provide multiple arrays in a hierarchy. At the lowest level of this hierarchy, each array element represents one unit of time. At the second level, each array element represents N units of time, where N is the number of elements in each array. At the third level, each array element represents N^2 units of time, and so on up the hierarchy. This approach allows the individual arrays to be indexed by different bits, as illustrated by Figure 14.9 for an unrealistically small eight-bit clock. Here, each array has 16 elements, so the low-order four bits of the time (currently `0xf`)

[8] At least assuming reasonably low packet-loss rates!

www.dbooks.org

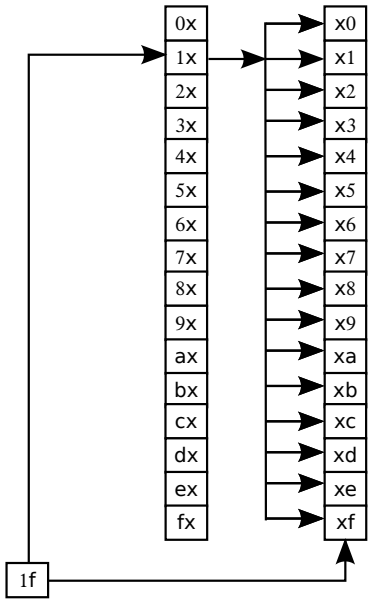


Figure 14.9: Timer Wheel

index the low-order (rightmost) array, and the next four bits (currently `0x1`) index the next level up. Thus, we have two arrays each with 16 elements, for a total of 32 elements, which, taken together, is much smaller than the 256-element array that would be required for a single array.

This approach works extremely well for throughput-based systems. Each timer operation is $O(1)$ with small constant, and each timer element is touched at most $m + 1$ times, where m is the number of levels.

Unfortunately, timer wheels do not work well for real-time systems, and for two reasons. The first reason is that there is a harsh tradeoff between timer accuracy and timer overhead, which is fancifully illustrated by Figures 14.10 and 14.11. In Figure 14.10, timer processing happens only once per millisecond, which keeps overhead acceptably low for many (but not all!) workloads, but which also means that timeouts cannot be set for finer than one-millisecond granularities. On the other hand, Figure 14.11 shows timer processing taking place every ten microseconds, which provides acceptably fine timer granularity for most (but not all!) workloads, but which processes timers so frequently that the system might well not have time to do anything else.

The second reason is the need to cascade timers from higher levels to lower levels. Referring back to Figure 14.9, we can see that any timers enqueued on element `1x` in



Figure 14.10: Timer Wheel at 1 kHz



Figure 14.11: Timer Wheel at 100 kHz

the upper (leftmost) array must be cascaded down to the lower (rightmost) array so that may be invoked when their time arrives. Unfortunately, there could be a large number of timeouts waiting to be cascaded, especially for timer wheels with larger numbers of levels. The power of statistics causes this cascading to be a non-problem for throughput-oriented systems, but cascading can result in problematic degradations of latency in real-time systems.

Of course, real-time systems could simply choose a different data structure, for example, some form of heap or tree, giving up $O(1)$ bounds on insertion and deletion operations to gain $O(\log n)$ limits on data-structure-maintenance operations. This can be a good choice for special-purpose RTOSes, but is inefficient for general-

purpose systems such as Linux, which routinely support extremely large numbers of timers.

The solution chosen for the Linux kernel's -rt patchset is to differentiate between timers that schedule later activity and timeouts that schedule error handling for low-probability errors such as TCP packet losses. One key observation is that error handling is normally not particularly time-critical, so that a timer wheel's millisecond-level granularity is good and sufficient. Another key observation is that error-handling timeouts are normally canceled very early, often before they can be cascaded. In addition, systems commonly have many more error-handling timeouts than they do timer events, so that an $O(\log n)$ data structure should provide acceptable performance for timer events.

However, it is possible to do better, namely by simply refusing to cascade timers. Instead of cascading, the timers that would otherwise have been cascaded all the way down the calendar queue are handled in place. This does result in up to a few percent error for the time duration, but the few situations where this is a problem can instead use tree-based high-resolution timers (hrtimers).

In short, the Linux kernel's -rt patchset uses timer wheels for error-handling timeouts and a tree for timer events, providing each category the required quality of service.

Threaded interrupts are used to address a significant source of degraded real-time latencies, namely long-running interrupt handlers, as shown in Figure 14.12. These latencies can be especially problematic for devices that can deliver a large number of events with a single interrupt, which means that the interrupt handler will run for an extended period of time processing all of these events. Worse yet are devices that can deliver new events to a still-running interrupt handler, as such an interrupt handler might well run indefinitely, thus indefinitely degrading real-time latencies.

One way of addressing this problem is the use of threaded interrupts shown in Figure 14.13. Interrupt handlers run in the context of a preemptible IRQ thread, which runs at a configurable priority. The device interrupt handler then runs for only a short time, just long enough to make the IRQ thread aware of the new event. As shown in the figure, threaded interrupts can greatly improve real-time latencies, in part because interrupt handlers running in the context of the IRQ thread may be preempted by high-priority real-time threads.

However, there is no such thing as a free lunch, and there are downsides to threaded interrupts. One downside is increased interrupt latency. Instead of immediately running the interrupt handler, the handler's execution is deferred until the IRQ thread gets around to running it. Of course, this is not a problem unless the device generating the interrupt is on the real-time application's critical path.

Another downside is that poorly written high-priority real-time code might starve the interrupt handler, for example, preventing networking code from running, in turn making it very difficult to debug the problem. Developers must therefore take great care when writing high-priority real-time code. This has been dubbed the *Spiderman principle*: With great power comes great responsibility.

Priority inheritance is used to handle priority inversion, which can be caused by, among other things, locks acquired by preemptible interrupt handlers [SRL90]. Suppose that a low-priority thread holds a lock, but is preempted by a group of medium-priority threads, at least one such thread per CPU. If an interrupt occurs, a high-priority IRQ thread will preempt one of the medium-priority threads, but only until it decides to acquire the lock held by the low-priority thread. Unfortunately, the low-priority thread cannot release the lock until it starts running, which the medium-priority threads prevent it from doing. So the high-priority IRQ thread cannot acquire the lock until after one of the medium-priority threads releases its CPU. In short, the medium-priority threads are indirectly blocking the high-priority IRQ threads, a classic case of priority inversion.

Note that this priority inversion could not happen with non-threaded interrupts because the low-priority thread would have to disable interrupts while holding the lock, which would prevent the medium-priority threads from preempting it.

In the priority-inheritance solution, the high-priority thread attempting to acquire the lock donates its priority to the low-priority thread holding the lock until such time as the lock is released, thus preventing long-term priority inversion.

Of course, priority inheritance does have its limitations. For example, if you can design your application to avoid priority inversion entirely, you will likely obtain somewhat better latencies [Yod04b]. This should be no surprise, given that priority inheritance adds a pair of context switches to the worst-case latency. That said, priority inheritance can convert indefinite postponement into a limited increase in latency, and the software-engineering

www.dbooks.org

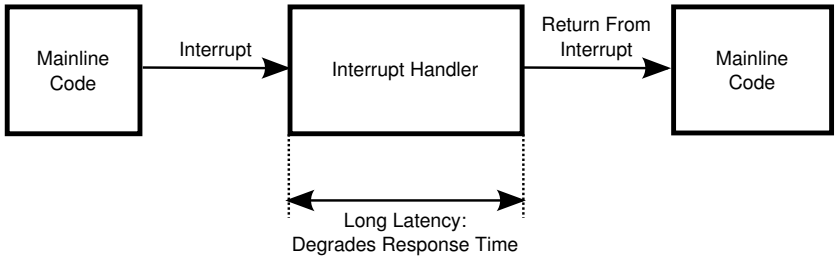


Figure 14.12: Non-Threaded Interrupt Handler

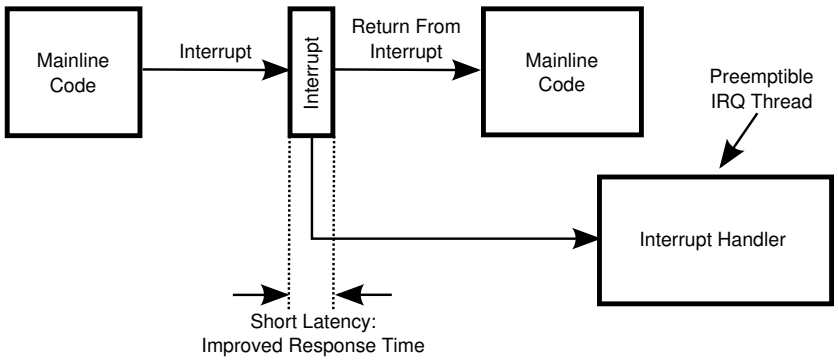


Figure 14.13: Threaded Interrupt Handler



Figure 14.14: Priority Inversion and User Input

benefits of priority inheritance may outweigh its latency costs in many applications.

Another limitation is that it addresses only lock-based priority inversions within the context of a given operating system. One priority-inversion scenario that it cannot address is a high-priority thread waiting on a network socket for a message that is to be written by a low-priority process that is preempted by a set of CPU-bound medium-priority processes. In addition, a potential disadvantage of applying priority inheritance to user input is fancifully depicted in Figure 14.14.

A final limitation involves reader-writer locking. Suppose that we have a very large number of low-priority threads, perhaps even thousands of them, each of which read-holds a particular reader-writer lock. Suppose that all of these threads are preempted by a set of medium-priority threads, with at least one medium-priority thread per CPU. Finally, suppose that a high-priority thread awakens and attempts to write-acquire this same reader-writer lock. No matter how vigorously we boost the priority of the threads read-holding this lock, it could well be a good long time before the high-priority thread can complete its write-acquisition.

There are a number of possible solutions to this reader-writer lock priority-inversion conundrum:

1. Only allow one read-acquisition of a given reader-writer lock at a time. (This is the approach traditionally taken by the Linux kernel's -rt patchset.)

2. Only allow N read-acquisitions of a given reader-writer lock at a time, where N is the number of CPUs.

3. Only allow *N* read-acquisitions of a given reader-writer lock at a time, where *N* is a number specified somehow by the developer. There is a good chance that the Linux kernel's -rt patchset will someday take this approach.

4. Prohibit high-priority threads from write-acquiring reader-writer locks that are ever read-acquired by threads running at lower priorities. (This is a variant of the *priority ceiling* protocol [SRL90].)

Quick Quiz 14.6: But if you only allow one reader at a time to read-acquire a reader-writer lock, isn't that the same as an exclusive lock??? ■

The no-concurrent-readers restriction eventually became intolerable, so the -rt developers looked more carefully at how the Linux kernel uses reader-writer spinlocks. They learned that time-critical code rarely uses those parts of the kernel that write-acquire reader-writer locks, so that the prospect of writer starvation was not a show-stopper. They therefore constructed a real-time reader-writer lock in which write-side acquisitions use priority inheritance among each other, but where read-side acquisitions take absolute priority over write-side acquisitions. This approach appears to be working well in practice, and is another lesson in the importance of clearly understanding what your users really need.

One interesting detail of this implementation is that both the `rt_read_lock()` and the `rt_write_lock()` functions enter an RCU read-side critical section and both the `rt_read_unlock()` and the `rt_write_unlock()` functions exit that critical section. This is necessary because non-realtime kernels' reader-writer locking functions disable preemption across their critical sections, and there really are reader-writer locking use cases that rely on the fact that `synchronize_rcu()` will therefore wait for all pre-exiting reader-writer-lock critical sections to complete. Let this be a lesson to you: Understanding what your users really need is critically important to correct operation, not just to performance. Not only that, but what your users really need changes over time.

This has the side-effect that all of a -rt kernel's reader-writer locking critical sections are subject to RCU priority boosting. This provides at least a partial solution to the problem of reader-writer lock readers being preempted for extended periods of time.

It is also possible to avoid reader-writer lock priority inversion by converting the reader-writer lock to RCU, as briefly discussed in the next section.

Listing 14.3: Preemptible Linux-Kernel RCU

```
1  void __rcu_read_lock(void)
2  {
3    current->rcu_read_lock_nesting++;
4    barrier();
5  }
6
7  void __rcu_read_unlock(void)
8  {
9    barrier();
10   if (!--current->rcu_read_lock_nesting)
11     barrier();
12     if (READ_ONCE(current->rcu_read_unlock_special.s)) {
13       rcu_read_unlock_special(t);
14     }
15 }
```

Preemptible RCU can sometimes be used as a replacement for reader-writer locking [MW07, MBWW12, McK14d], as was discussed in Section 9.5. Where it can be used, it permits readers and updaters to run concurrently, which prevents low-priority readers from inflicting any sort of priority-inversion scenario on high-priority updaters. However, for this to be useful, it is necessary to be able to preempt long-running RCU read-side critical sections [GMTW08]. Otherwise, long RCU read-side critical sections would result in excessive real-time latencies.

A preemptible RCU implementation was therefore added to the Linux kernel. This implementation avoids the need to individually track the state of each and every task in the kernel by keeping lists of tasks that have been preempted within their current RCU read-side critical sections. A grace period is permitted to end: (1) Once all CPUs have completed any RCU read-side critical sections that were in effect before the start of the current grace period and (2) Once all tasks that were preempted while in one of those pre-existing critical sections have removed themselves from their lists. A simplified version of this implementation is shown in Listing 14.3. The `__rcu_read_lock()` function spans lines 1–5 and the `__rcu_read_unlock()` function spans lines 7–15.

Line 3 of `__rcu_read_lock()` increments a per-task count of the number of nested `rcu_read_lock()` calls, and line 4 prevents the compiler from reordering the subsequent code in the RCU read-side critical section to precede the `rcu_read_lock()`.

Line 9 of `__rcu_read_unlock()` prevents the compiler from reordering the code in the critical section with the remainder of this function. Line 10 decrements the nesting count and checks to see if it has become zero, in other words, if this corresponds to the outermost `rcu_read_unlock()` of a nested set. If so, line 11 prevents the compiler from reordering this nesting update with

www.dbooks.org

line 12's check for special handling. If special handling is required, then the call to `rcu_read_unlock_special()` on 13 carries it out.

There are several types of special handling that can be required, but we will focus on that required when the RCU read-side critical section has been preempted. In this case, the task must remove itself from the list that it was added to when it was first preempted within its RCU read-side critical section. However, it is important to note that these lists are protected by locks, which means that `rcu_read_unlock()` is no longer lockless. However, the highest-priority threads will not be preempted, and therefore, for those highest-priority threads, `rcu_read_unlock()` will never attempt to acquire any locks. In addition, if implemented carefully, locking can be used to synchronize real-time software [Bra11, SM04a].

Quick Quiz 14.7: Suppose that preemption occurs just after the load from `t->rcu_read_unlock_special.s` on line 12 of Listing 14.3. Mightn't that result in the task failing to invoke `rcu_read_unlock_special()`, thus failing to remove itself from the list of tasks blocking the current grace period, in turn causing that grace period to extend indefinitely? ■

Another important real-time feature of RCU, whether preemptible or not, is the ability to offload RCU callback execution to a kernel thread. To use this, your kernel must be built with `CONFIG_RCU_NOCB_CPU=y` and booted with the `rcu_nocbs=` kernel boot parameter specifying which CPUs are to be offloaded. Alternatively, any CPU specified by the `nohz_full=` kernel boot parameter described in Section 14.3.5.2 will also have its RCU callbacks offloaded.

In short, this preemptible RCU implementation enables real-time response for read-mostly data structures without the delays inherent to priority boosting of large numbers of readers, and also without delays due to callback invocation.

Preemptible spinlocks are an important part of the -rt patchset due to the long-duration spinlock-based critical sections in the Linux kernel. This functionality has not yet reached mainline: Although they are a conceptually simple substitution of sleeplocks for spinlocks, they have proven relatively controversial. In addition the real-time functionality that is already in the mainline Linux kernel suffices for a great many use cases, which slowed the -rt patchset's development rate in the early 2010s [Edg13, Edg14]. However, preemptible spinlocks are absolutely necessary to the task of achieving real-time latencies down in the tens of microseconds. Fortunately, Linux Foundation organized an effort to fund moving the remaining code from the -rt patchset to mainline.

Per-CPU variables are used heavily in the Linux kernel for performance reasons. Unfortunately for real-time applications, many use cases for per-CPU variables require coordinated update of multiple such variables, which is normally provided by disabling preemption, which in turn degrades real-time latencies. Real-time applications clearly need some other way of coordinating per-CPU variable updates.

One alternative is to supply per-CPU spinlocks, which as noted above are actually sleeplocks, so that their critical sections can be preempted and so that priority inheritance is provided. In this approach, code updating groups of per-CPU variables must acquire the current CPU's spinlock, carry out the update, then release whichever lock is acquired, keeping in mind that a preemption might have resulted in a migration to some other CPU. However, this approach introduces both overhead and deadlocks.

Another alternative, which is used in the -rt patchset as of early 2021, is to convert preemption disabling to migration disabling. This ensures that a given kernel thread remains on its CPU through the duration of the per-CPU-variable update, but could also allow some other kernel thread to intersperse its own update of those same variables, courtesy of preemption. There are cases such as statistics gathering where this is not a problem. In the surprisingly rare case where such mid-update preemption is a problem, the use case at hand must properly synchronize the updates, perhaps through a set of per-CPU locks specific to that use case. Although introducing locks again introduces the possibility of deadlock, the per-use-case nature of these locks makes any such deadlocks easier to manage and avoid.

Closing event-driven remarks. There are of course any number of other Linux-kernel components that are critically important to achieving world-class real-time latencies, for example, deadline scheduling [dO18b, dO18a], however, those listed in this section give a good feeling for the workings of the Linux kernel augmented by the -rt patchset.

14.3.5.2 Polling-Loop Real-Time Support

At first glance, use of a polling loop might seem to avoid all possible operating-system interference problems. After all, if a given CPU never enters the kernel, the kernel

is completely out of the picture. And the traditional approach to keeping the kernel out of the way is simply not to have a kernel, and many real-time applications do indeed run on bare metal, particularly those running on eight-bit microcontrollers.

One might hope to get bare-metal performance on a modern operating-system kernel simply by running a single CPU-bound user-mode thread on a given CPU, avoiding all causes of interference. Although the reality is of course more complex, it is becoming possible to do just that, courtesy of the `NO_HZ_FULL` implementation led by Frederic Weisbecker [Cor13, Wei12] that was accepted into version 3.10 of the Linux kernel. Nevertheless, considerable care is required to properly set up such an environment, as it is necessary to control a number of possible sources of OS jitter. The discussion below covers the control of several sources of OS jitter, including device interrupts, kernel threads and daemons, scheduler real-time throttling (this is a feature, not a bug!), timers, non-real-time device drivers, in-kernel global synchronization, scheduling-clock interrupts, page faults, and finally, non-real-time hardware and firmware.

Interrupts are an excellent source of large amounts of OS jitter. Unfortunately, in most cases interrupts are absolutely required in order for the system to communicate with the outside world. One way of resolving this conflict between OS jitter and maintaining contact with the outside world is to reserve a small number of housekeeping CPUs, and to force all interrupts to these CPUs. The `Documentation/IRQ-affinity.txt` file in the Linux source tree describes how to direct device interrupts to specified CPU, which as of early 2021 involves something like the following:

```
$ echo 0f > /proc/irq/44/smp_affinity
```

This command would confine interrupt #44 to CPUs 0–3. Note that scheduling-clock interrupts require special handling, and are discussed later in this section.

A second source of OS jitter is due to kernel threads and daemons. Individual kernel threads, such as RCU's grace-period kthreads (`rcu_bh`, `rcu_preempt`, and `rcu_sched`), may be forced onto any desired CPUs using the `taskset` command, the `sched_setaffinity()` system call, or `cgroups`.

Per-CPU kthreads are often more challenging, sometimes constraining hardware configuration and workload layout. Preventing OS jitter from these kthreads requires either that certain types of hardware not be attached to real-time systems, that all interrupts and I/O initiation take place on housekeeping CPUs, that special kernel Kconfig or boot parameters be selected in order to direct work away from the worker CPUs, or that worker CPUs never enter the kernel. Specific per-kthread advice may be found in the Linux kernel source `Documentation` directory at `kernel-per-CPU-kthreads.txt`.

A third source of OS jitter in the Linux kernel for CPU-bound threads running at real-time priority is the scheduler itself. This is an intentional debugging feature, designed to ensure that important non-realtime work is allotted at least 50 milliseconds out of each second, even if there is an infinite-loop bug in your real-time application. However, when you are running a polling-loop-style real-time application, you will need to disable this debugging feature. This can be done as follows:

```
$ echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

You will of course need to be running as root to execute this command, and you will also need to carefully consider the aforementioned Spiderman principle. One way to minimize the risks is to offload interrupts and kernel threads/daemons from all CPUs running CPU-bound real-time threads, as described in the paragraphs above. In addition, you should carefully read the material in the `Documentation/scheduler` directory. The material in the `sched-rt-group.rst` file is particularly important, especially if you are using the `cgroups` real-time features enabled by the `CONFIG_RT_GROUP_SCHED` Kconfig parameter.

A fourth source of OS jitter comes from timers. In most cases, keeping a given CPU out of the kernel will prevent timers from being scheduled on that CPU. One important exception are recurring timers, where a given timer handler posts a later occurrence of that same timer. If such a timer gets started on a given CPU for any reason, that timer will continue to run periodically on that CPU, inflicting OS jitter indefinitely. One crude but effective way to offload recurring timers is to use CPU hotplug to offline all worker CPUs that are to run CPU-bound real-time application threads, online these same CPUs, then start your real-time application.

A fifth source of OS jitter is provided by device drivers that were not intended for real-time use. For an old canonical example, in 2005, the VGA driver would blank the screen by zeroing the frame buffer with interrupts disabled, which resulted in tens of milliseconds of OS jitter. One way of avoiding device-driver-induced OS jitter is to carefully select devices that have been used heavily in real-time systems, and which have therefore

www.dbooks.org

had their real-time bugs fixed. Another way is to confine the device's interrupts and all code using that device to designated housekeeping CPUs. A third way is to test the device's ability to support real-time workloads and fix any real-time bugs.[9]

A sixth source of OS jitter is provided by some in-kernel full-system synchronization algorithms, perhaps most notably the global TLB-flush algorithm. This can be avoided by avoiding memory-unmapping operations, and especially avoiding unmapping operations within the kernel. As of early 2021, the way to avoid in-kernel unmapping operations is to avoid unloading kernel modules.

A seventh source of OS jitter is provided by scheduling-clock interrrupts and RCU callback invocation. These may be avoided by building your kernel with the `NO_HZ_FULL` Kconfig parameter enabled, and then booting with the `nohz_full=` parameter specifying the list of worker CPUs that are to run real-time threads. For example, `nohz_full=2-7` would designate CPUs 2, 3, 4, 5, 6, and 7 as worker CPUs, thus leaving CPUs 0 and 1 as housekeeping CPUs. The worker CPUs would not incur scheduling-clock interrupts as long as there is no more than one runnable task on each worker CPU, and each worker CPU's RCU callbacks would be invoked on one of the housekeeping CPUs. A CPU that has suppressed scheduling-clock interrupts due to there only being one runnable task on that CPU is said to be in *adaptive ticks mode* or in `nohz_full` mode. It is important to ensure that you have designated enough housekeeping CPUs to handle the housekeeping load imposed by the rest of the system, which requires careful benchmarking and tuning.

An eighth source of OS jitter is page faults. Because most Linux implementations use an MMU for memory protection, real-time applications running on these systems can be subject to page faults. Use the `mlock()` and `mlockall()` system calls to pin your application's pages into memory, thus avoiding major page faults. Of course, the Spiderman principle applies, because locking down too much memory may prevent the system from getting other work done.

A ninth source of OS jitter is unfortunately the hardware and firmware. It is therefore important to use systems that have been designed for real-time use.

Unfortunately, this list of OS-jitter sources can never be complete, as it will change with each new version of the kernel. This makes it necessary to be able to track down additional sources of OS jitter. Given a CPU *N* running a CPU-bound usermode thread, the commands shown in Listing 14.4 will produce a list of all the times that this CPU entered the kernel. Of course, the `N` on line 5 must be replaced with the number of the CPU in question, and the `1` on line 2 may be increased to show additional levels of function call within the kernel. The resulting trace can help track down the source of the OS jitter.

Listing 14.4: Locating Sources of OS Jitter

```
1 cd /sys/kernel/debug/tracing
2 echo 1 > max_graph_depth
3 echo function_graph > current_tracer
4 # run workload
5 cat per_cpu/cpuN/trace
```

As always, there is no free lunch, and `NO_HZ_FULL` is no exception. As noted earlier, `NO_HZ_FULL` makes kernel/user transitions more expensive due to the need for delta process accounting and the need to inform kernel subsystems (such as RCU) of the transitions. As a rough rule of thumb, `NO_HZ_FULL` helps with many types of real-time and heavy-compute workloads, but hurts other workloads that feature high rates of system calls and I/O [ACA+18]. Additional limitations, tradeoffs, and configuration advice may be found in `Documentation/timers/no_hz.rst`.

As you can see, obtaining bare-metal performance when running CPU-bound real-time threads on a general-purpose OS such as Linux requires painstaking attention to detail. Automation would of course help, and some automation has been applied, but given the relatively small number of users, automation can be expected to appear relatively slowly. Nevertheless, the ability to gain near-bare-metal performance while running a general-purpose operating system promises to ease construction of some types of real-time systems.

14.3.6 Implementing Parallel Real-Time Applications

Developing real-time applications is a wide-ranging topic, and this section can only touch on a few aspects. To this end, Section 14.3.6.1 looks at a few software components commonly used in real-time applications, Section 14.3.6.2 provides a brief overview of how polling-loop-based applications may be implemented, Section 14.3.6.3 gives a similar overview of streaming applications, and Section 14.3.6.4 briefly covers event-based applications.

[9] If you take this approach, please submit your fixes upstream so that others can benefit. After all, when you need to port your application to a later version of the Linux kernel, *you* will be one of those "others".

14.3.6.1 Real-Time Components

As in all areas of engineering, a robust set of components is essential to productivity and reliability. This section is not a full catalog of real-time software components—such a catalog would fill multiple books—but rather a brief overview of the types of components available.

A natural place to look for real-time software components would be algorithms offering wait-free synchronization [Her91], and in fact lockless algorithms are very important to real-time computing. However, wait-free synchronization only guarantees forward progress in finite time. Although a century is finite, this is unhelpful when your deadlines are measured in microseconds, let alone milliseconds.

Nevertheless, there are some important wait-free algorithms that do provide bounded response time, including atomic test and set, atomic exchange, atomic fetch-and-add, single-producer/single-consumer FIFO queues based on circular arrays, and numerous per-thread partitioned algorithms. In addition, recent research has confirmed the observation that algorithms with lock-free guarantees[10] also provide the same latencies in practice (in the wait-free sense), assuming a stochastically fair scheduler and absence of fail-stop bugs [ACHS13]. This means that many non-wait-free stacks and queues are nevertheless appropriate for real-time use.

Quick Quiz 14.8: But isn't correct operation despite fail-stop bugs a valuable fault-tolerance property? ■

In practice, locking is often used in real-time programs, theoretical concerns notwithstanding. However, under more severe constraints, lock-based algorithms can also provide bounded latencies [Bra11]. These constraints include:

1. Fair scheduler. In the common case of a fixed-priority scheduler, the bounded latencies are provided only to the highest-priority threads.
2. Sufficient bandwidth to support the workload. An implementation rule supporting this constraint might be "There will be at least 50 % idle time on all CPUs during normal operation," or, more formally, "The offered load will be sufficiently low to allow the workload to be schedulable at all times."
3. No fail-stop bugs.
4. FIFO locking primitives with bounded acquisition, handoff, and release latencies. Again, in the common case of a locking primitive that is FIFO within priorities, the bounded latencies are provided only to the highest-priority threads.
5. Some way of preventing unbounded priority inversion. The priority-ceiling and priority-inheritance disciplines mentioned earlier in this chapter suffice.
6. Bounded nesting of lock acquisitions. We can have an unbounded number of locks, but only as long as a given thread never acquires more than a few of them (ideally only one of them) at a time.
7. Bounded number of threads. In combination with the earlier constraints, this constraint means that there will be a bounded number of threads waiting on any given lock.
8. Bounded time spent in any given critical section. Given a bounded number of threads waiting on any given lock and a bounded critical-section duration, the wait time will be bounded.

Quick Quiz 14.9: I couldn't help but spot the word "includes" before this list. Are there other constraints? ■

This result opens a vast cornucopia of algorithms and data structures for use in real-time software—and validates long-standing real-time practice.

Of course, a careful and simple application design is also extremely important. The best real-time components in the world cannot make up for a poorly thought-out design. For parallel real-time applications, synchronization overheads clearly must be a key component of the design.

14.3.6.2 Polling-Loop Applications

Many real-time applications consist of a single CPU-bound loop that reads sensor data, computes a control law, and writes control output. If the hardware registers providing sensor data and taking control output are mapped into the application's address space, this loop might be completely free of system calls. But beware of the Spiderman principle: With great power comes great responsibility, in this case the responsibility to avoid bricking the hardware by making inappropriate references to the hardware registers.

This arrangement is often run on bare metal, without the benefits of (or the interference from) an operating

[10] Wait-free algorithms guarantee that all threads make progress in finite time, while lock-free algorithms only guarantee that at least one thread will make progress in finite time. See Section 14.2 for more details.

www.dbooks.org

system. However, increasing hardware capability and increasing levels of automation motivates increasing software functionality, for example, user interfaces, logging, and reporting, all of which can benefit from an operating system.

One way of gaining much of the benefit of running on bare metal while still having access to the full features and functions of a general-purpose operating system is to use the Linux kernel's `NO_HZ_FULL` capability, described in Section 14.3.5.2.

Listing 14.5: Timed-Wait Test Program

```
1 if (clock_gettime(CLOCK_REALTIME, &timestart) != 0) {
2   perror("clock_gettime 1");
3   exit(-1);
4 }
5 if (nanosleep(&timewait, NULL) != 0) {
6   perror("nanosleep");
7   exit(-1);
8 }
9 if (clock_gettime(CLOCK_REALTIME, &timeend) != 0) {
10  perror("clock_gettime 2");
11  exit(-1);
12 }
```

14.3.6.3 Streaming Applications

One type of big-data real-time application takes input from numerous sources, processes it internally, and outputs alerts and summaries. These *streaming applications* are often highly parallel, processing different information sources concurrently.

One approach for implementing streaming applications is to use dense-array circular FIFOs to connect different processing steps [Sut13]. Each such FIFO has only a single thread producing into it and a (presumably different) single thread consuming from it. Fan-in and fan-out points use threads rather than data structures, so if the output of several FIFOs needed to be merged, a separate thread would input from them and output to another FIFO for which this separate thread was the sole producer. Similarly, if the output of a given FIFO needed to be split, a separate thread would input from this FIFO and output to several FIFOs as needed.

This discipline might seem restrictive, but it allows communication among threads with minimal synchronization overhead, and minimal synchronization overhead is important when attempting to meet tight latency constraints. This is especially true when the amount of processing for each step is small, so that the synchronization overhead is significant compared to the processing overhead.

The individual threads might be CPU-bound, in which case the advice in Section 14.3.6.2 applies. On the other hand, if the individual threads block waiting for data from their input FIFOs, the advice of the next section applies.

14.3.6.4 Event-Driven Applications

We will use fuel injection into a mid-sized industrial engine as a fanciful example for event-driven applications. Under normal operating conditions, this engine requires that the fuel be injected within a one-degree interval surrounding top dead center. If we assume a 1,500-RPM rotation rate, we have 25 rotations per second, or about 9,000 degrees of rotation per second, which translates to 111 microseconds per degree. We therefore need to schedule the fuel injection to within a time interval of about 100 microseconds.

Suppose that a timed wait was to be used to initiate fuel injection, although if you are building an engine, I hope you supply a rotation sensor. We need to test the timed-wait functionality, perhaps using the test program shown in Listing 14.5. Unfortunately, if we run this program, we can get unacceptable timer jitter, even in a -rt kernel.

One problem is that POSIX `CLOCK_REALTIME` is, oddly enough, not intended for real-time use. Instead, it means "realtime" as opposed to the amount of CPU time consumed by a process or thread. For real-time use, you should instead use `CLOCK_MONOTONIC`. However, even with this change, results are still unacceptable.

Another problem is that the thread must be raised to a real-time priority by using the `sched_setscheduler()` system call. But even this change is insufficient, because we can still see page faults. We also need to use the `mlockall()` system call to pin the application's memory, preventing page faults. With all of these changes, results might finally be acceptable.

In other situations, further adjustments might be needed. It might be necessary to affinity time-critical threads onto their own CPUs, and it might also be necessary to affinity interrupts away from those CPUs. It might be necessary to carefully select hardware and drivers, and it will very likely be necessary to carefully select kernel configuration.

As can be seen from this example, real-time computing can be quite unforgiving.

14.3.6.5 The Role of RCU

Suppose that you are writing a parallel real-time application that needs to access data that is subject to gradual change, perhaps due to changes in temperature, humidity, and barometric pressure. The real-time response constraints on this program are so severe that it is not

Listing 14.6: Real-Time Calibration Using RCU

```
1  struct calibration {
2    short a;
3    short b;
4    short c;
5  };
6  struct calibration default_cal = { 62, 33, 88 };
7  struct calibration cur_cal = &default_cal;
8
9  short calc_control(short t, short h, short press)
10 {
11   struct calibration *p;
12
13   p = rcu_dereference(cur_cal);
14   return do_control(t, h, press, p->a, p->b, p->c);
15 }
16
17 bool update_cal(short a, short b, short c)
18 {
19   struct calibration *p;
20   struct calibration *old_p;
21
22   old_p = rcu_dereference(cur_cal);
23   p = malloc(sizeof(*p);
24   if (!p)
25     return false;
26   p->a = a;
27   p->b = b;
28   p->c = c;
29   rcu_assign_pointer(cur_cal, p);
30   if (old_p == &default_cal)
31     return true;
32   synchronize_rcu();
33   free(p);
34   return true;
35 }
```

permissible to spin or block, thus ruling out locking, nor is it permissible to use a retry loop, thus ruling out sequence locks and hazard pointers. Fortunately, the temperature and pressure are normally controlled, so that a default hard-coded set of data is usually sufficient.

However, the temperature, humidity, and pressure occasionally deviate too far from the defaults, and in such situations it is necessary to provide data that replaces the defaults. Because the temperature, humidity, and pressure change gradually, providing the updated values is not a matter of urgency, though it must happen within a few minutes. The program is to use a global pointer imaginatively named `cur_cal` that normally references `default_cal`, which is a statically allocated and initialized structure that contains the default calibration values in fields imaginatively named `a`, `b`, and `c`. Otherwise, `cur_cal` points to a dynamically allocated structure providing the current calibration values.

Listing 14.6 shows how RCU can be used to solve this problem. Lookups are deterministic, as shown in `calc_control()` on lines 9–15, consistent with real-time requirements. Updates are more complex, as shown by `update_cal()` on lines 17–35.



Figure 14.15: The Dark Side of Real-Time Computing



Figure 14.16: The Dark Side of Real-Fast Computing

Quick Quiz 14.10: Given that real-time systems are often used for safety-critical applications, and given that runtime memory allocation is forbidden in many safety-critical situations, what is with the call to `malloc()`??? ■

Quick Quiz 14.11: Don't you need some kind of synchronization to protect `update_cal()`? ■

This example shows how RCU can provide deterministic read-side data-structure access to real-time programs.

14.3.7 Real Time vs. Real Fast: How to Choose?

The choice between real-time and real-fast computing can be a difficult one. Because real-time systems often inflict

www.dbooks.org

a throughput penalty on non-real-time computing, using real-time when it is not required is unwise, as fancifully depicted by Figure 14.15. On the other hand, failing to use real-time when it *is* required can also cause problems, as fancifully depicted by Figure 14.16. It is almost enough to make you feel sorry for the boss!

One rule of thumb uses the following four questions to help you choose:

1. Is average long-term throughput the only goal?
2. Is it permissible for heavy loads to degrade response times?
3. Is there high memory pressure, ruling out use of the `mlockall()` system call?
4. Does the basic work item of your application take more than 100 milliseconds to complete?

If the answer to any of these questions is "yes", you should choose real-fast over real-time, otherwise, real-time might be for you.

Choose wisely, and if you do choose real-time, make sure that your hardware, firmware, and operating system are up to the job!

The art of progress is to preserve order amid change and to preserve change amid order.

Alfred North Whitehead

Chapter 15

Advanced Synchronization: Memory Ordering

Causality and sequencing are deeply intuitive, and hackers often tend to have a much stronger grasp of these concepts than does the general population. These intuitions can be extremely powerful tools when writing, analyzing, and debugging both sequential code and parallel code that makes use of standard mutual-exclusion mechanisms, especially locking. Unfortunately, these intuitions break down completely in face of code that fails to use standard mechanisms, one important example of course being the code that implements these standard mechanisms, and another being performance-critical code that uses weaker synchronization. Nevertheless, some argue that weakness is a virtue [Alg13]. This chapter will help you gain an understanding of memory ordering, that, with practice, will be sufficient to implement synchronization primitives and performance-critical code.

Section 15.1 will demonstrate that real computer systems can reorder memory references, give some reasons why they do so, and provide some information on how to prevent undesired reordering. Sections 15.2 and 15.3 will provide a more complete list of things that hardware and compilers, respectively, can do to unwary parallel programmers. Section 15.4 gives an overview of the benefits of modeling memory ordering at higher levels of abstraction. Section 15.5 follows up with more detail on a few representative hardware platforms. Finally, Section 15.6 provides some useful rules of thumb.

Quick Quiz 15.1: This chapter has been rewritten since the first edition. Did memory ordering change all *that* since 2014? ■

15.1 Ordering: Why and How?

Nothing is orderly till people take hold of it.
Everything in creation lies around loose.

Henry Ward Beecher, updated

One motivation for memory ordering can be seen in the trivial-seeming litmus test in Listing 15.1 (`C-SB+o-o+o-o.litmus`), which at first glance might appear to guarantee that the `exists` clause never triggers.[1] After all, if `0:r2=0` as shown in the `exists` clause,[2] we might hope that Thread `P0()`'s load from `x1` into `r2` must have happened before Thread `P1()`'s store to `x1`, which might raise further hopes that Thread `P1()`'s load from `x0` into `r2` must happen after Thread `P0()`'s store to `x0`, so that `1:r2=2`, thus not triggering the `exists` clause. The example is symmetric, so similar reasoning might lead us to hope that `1:r2=0` guarantees that `0:r2=2`. Unfortunately, the lack of memory barriers dashes these hopes. The CPU is within its rights to reorder the statements within both Thread `P0()` and Thread `P1()`, even on relatively strongly ordered systems such as x86.

Quick Quiz 15.2: The compiler can also reorder Thread `P0()`'s and Thread `P1()`'s memory accesses in Listing 15.1, right? ■

This willingness to reorder can be confirmed using tools such as `litmus7` [AMT14], which found that the counter-intuitive ordering happened 314 times out of 100,000,000

[1] Purists would instead insist that the `exists` clause is never *satisfied*, but we use "trigger" here by analogy with assertions.

[2] That is, Thread `P0()`'s instance of local variable `r2` equals zero. See Section 12.2.1 for documentation of litmus-test nomenclature.

www.dbooks.org

Listing 15.1: Memory Misordering: Store-Buffering Litmus Test

```
1  C C-SB+o-o+o-o
2
3  {}
4
5  P0(int *x0, int *x1)
6  {
7    int r2;
8
9    WRITE_ONCE(*x0, 2);
10   r2 = READ_ONCE(*x1);
11 }
12
13 P1(int *x0, int *x1)
14 {
15   int r2;
16
17   WRITE_ONCE(*x1, 2);
18   r2 = READ_ONCE(*x0);
19 }
20
21 exists (1:r2=0 /\ 0:r2=0)
```

trials on my x86 laptop. Oddly enough, the perfectly legal outcome where both loads return the value 2 occurred less frequently, in this case, only 167 times.[3] The lesson here is clear: Increased counter-intuitivity does not necessarily imply decreased probability!

The following sections show exactly where this intuition breaks down, and then puts forward a mental model of memory ordering that can help you avoid these pitfalls.

Section 15.1.1 gives a brief overview of why hardware misorders memory accesses, and then Section 15.1.2 gives an equally brief overview of how you can thwart such misordering. Finally, Section 15.1.3 lists some basic rules of thumb, which will be further refined in later sections. These sections focus on hardware reordering, but rest assured that compilers reorder much more aggressively than hardware ever dreamed of doing. But that topic will be taken up later in Section 15.3.

15.1.1 Why Hardware Misordering?

But why does memory misordering happen in the first place? Can't CPUs keep track of ordering on their own? Isn't that why we have computers in the first place, to keep track of things?

Many people do indeed expect their computers to keep track of things, but many also insist that they keep track of things quickly. However, as seen in Chapter 3, main memory cannot keep up with modern CPUs, which can execute hundreds of instructions in the time required to fetch a single variable from memory. CPUs therefore sport

[3] Please note that results are sensitive to the exact hardware configuration, how heavily the system is loaded, and much else besides.

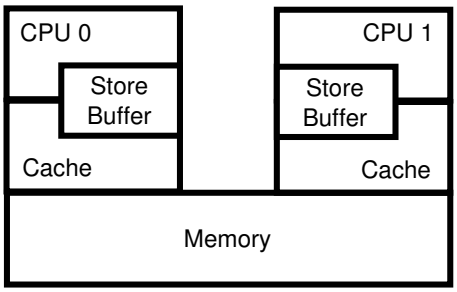


Figure 15.1: System Architecture With Store Buffers

increasingly large caches, as seen back in Figure 3.10, which means that although the first load by a given CPU from a given variable will result in an expensive *cache miss* as was discussed in Section 3.1.5, subsequent repeated loads from that variable by that CPU might execute very quickly because the initial cache miss will have loaded that variable into that CPU's cache.

However, it is also necessary to accommodate frequent concurrent stores from multiple CPUs to a set of shared variables. In cache-coherent systems, if the caches hold multiple copies of a given variable, all the copies of that variable must have the same value. This works extremely well for concurrent loads, but not so well for concurrent stores: Each store must do something about all copies of the old value (another cache miss!), which, given the finite speed of light and the atomic nature of matter, will be slower than impatient software hackers would like.

CPUs therefore come equipped with store buffers, as shown in Figure 15.1. When a given CPU stores to a variable not present in that CPU's cache, then the new value is instead placed in that CPU's store buffer. The CPU can then proceed immediately, without having to wait for the store to do something about all the old values of that variable residing in other CPUs' caches.

Although store buffers can greatly increase performance, they can cause instructions and memory references to execute out of order, which can in turn cause serious confusion, as fancifully illustrated in Figure 15.2.

In particular, store buffers cause the memory misordering illustrated by Listing 15.1. Table 15.1 shows the steps leading to this misordering. Row 1 shows the initial state, where CPU 0 has `x1` in its cache and CPU 1 has `x0` in its cache, both variables having a value of zero. Row 2 shows the state change due to each CPU's store (lines 9 and 17 of Listing 15.1). Because neither CPU has the stored-to

Table 15.1: Memory Misordering: Store-Buffering Sequence of Events

	CPU 0			CPU 1		
	Instruction	Store Buffer	Cache	Instruction	Store Buffer	Cache
1	(Initial state)		x1==0	(Initial state)		x0==0
2	x0 = 2;	x0==2	x1==0	x1 = 2;	x1==2	x0==0
3	r2 = x1; (0)	x0==2	x1==0	r2 = x0; (0)	x1==2	x0==0
4	(Read-invalidate)	x0==2	x0==0	(Read-invalidate)	x1==2	x1==0
5	(Finish store)		x0==2	(Finish store)		x1==2

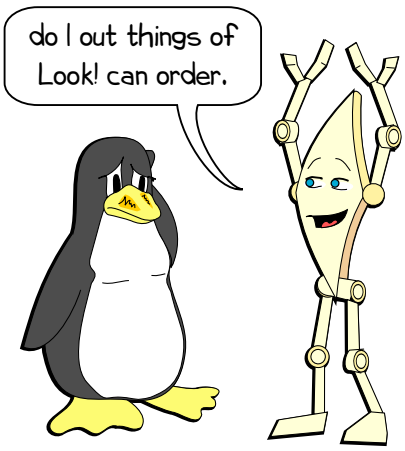


Figure 15.2: CPUs Can Do Things Out of Order

variable in its cache, both CPUs record their stores in their respective store buffers.

Quick Quiz 15.3: But wait!!! On row 2 of Table 15.1 both x0 and x1 each have two values at the same time, namely zero and two. How can that possibly work??? ■

Row 3 shows the two loads (lines 10 and 18 of Listing 15.1). Because the variable being loaded by each CPU is in that CPU's cache, each load immediately returns the cached value, which in both cases is zero.

But the CPUs are not done yet: Sooner or later, they must empty their store buffers. Because caches move data around in relatively large blocks called *cachelines*, and because each cacheline can hold several variables, each CPU must get the cacheline into its own cache so that it can update the portion of that cacheline corresponding to the variable in its store buffer, but without disturbing any other part of the cacheline. Each CPU must also ensure that the cacheline is not present in any other CPU's cache, for which a read-invalidate operation is used. As shown on row 4, after both read-invalidate operations complete, the two CPUs have traded cachelines, so that CPU 0's cache now contains x0 and CPU 1's cache now contains x1. Once these two variables are in their new homes, each CPU can flush its store buffer into the corresponding cache line, leaving each variable with its final value as shown on row 5.

Quick Quiz 15.4: But don't the values also need to be flushed from the cache to main memory? ■

In summary, store buffers are needed to allow CPUs to handle store instructions efficiently, but they can result in counter-intuitive memory misordering.

But what do you do if your algorithm really needs its memory references to be ordered? For example, suppose that you are communicating with a driver using a pair of flags, one that says whether or not the driver is running and the other that says whether there is a request pending for that driver. The requester needs to set the request-pending flag, then check the driver-running flag, and if false, wake the driver. Once the driver has serviced all the pending requests that it knows about, it needs to clear its driver-running flag, then check the request-pending flag to see if it needs to restart. This very reasonable approach cannot work unless there is some way to make sure that the hardware processes the stores and loads in order. This is the subject of the next section.

15.1.2 How to Force Ordering?

It turns out that there are compiler directives and synchronization primitives (such as locking and RCU) that are responsible for maintaining the illusion of ordering through use of *memory barriers* (for example, smp_mb() in the Linux kernel). These memory barriers can be explicit instructions, as they are on Arm, POWER, Itanium, and Alpha, or they can be implied by other instructions,

www.dbooks.org

Listing 15.2: Memory Ordering: Store-Buffering Litmus Test

```
1  C C-SB+o-mb-o+o-mb-o
2
3  {}
4
5  P0(int *x0, int *x1)
6  {
7    int r2;
8
9    WRITE_ONCE(*x0, 2);
10   smp_mb();
11   r2 = READ_ONCE(*x1);
12 }
13
14 P1(int *x0, int *x1)
15 {
16   int r2;
17
18   WRITE_ONCE(*x1, 2);
19   smp_mb();
20   r2 = READ_ONCE(*x0);
21 }
22
23 exists (1:r2=0 /\ 0:r2=0)
```

as they often are on x86. Since these standard synchronization primitives preserve the illusion of ordering, your path of least resistance is to simply use these primitives, thus allowing you to stop reading this section.

However, if you need to implement the synchronization primitives themselves, or if you are simply interested in understanding how memory ordering works, read on! The first stop on the journey is Listing 15.2 (`C-SB+o-mb-o+o-mb-o.litmus`), which places an `smp_mb()` Linux-kernel full memory barrier between the store and load in both `P0()` and `P1()`, but is otherwise identical to Listing 15.1. These barriers prevent the counter-intuitive outcome from happening on 100,000,000 trials on my x86 laptop. Interestingly enough, the added overhead due to these barriers causes the legal outcome where both loads return the value two to happen more than 800,000 times, as opposed to only 167 times for the barrier-free code in Listing 15.1.

These barriers have a profound effect on ordering, as can be seen in Table 15.2. Although the first two rows are the same as in Table 15.1 and although the `smp_mb()` instructions on row 3 do not change state in and of themselves, they do cause the stores to complete (rows 4 and 5) before the loads (row 6), which rules out the counter-intuitive outcome shown in Table 15.1. Note that variables `x0` and `x1` each still have more than one value on row 2, however, as promised earlier, the `smp_mb()` invocations straighten things out in the end.

Although full barriers such as `smp_mb()` have extremely strong ordering guarantees, their strength comes at a high price in terms of foregone hardware and compiler optimizations. A great many situations can be handled with much weaker ordering guarantees that use much cheaper memory-ordering instructions, or, in some case, no memory-ordering instructions at all.

Table 15.3 provides a cheatsheet of the Linux kernel's ordering primitives and their guarantees. Each row corresponds to a primitive or category of primitives that might or might not provide ordering, with the columns labeled "Prior Ordered Operation" and "Subsequent Ordered Operation" being the operations that might (or might not) be ordered against. Cells containing "Y" indicate that ordering is supplied unconditionally, while other characters indicate that ordering is supplied only partially or conditionally. Blank cells indicate that no ordering is supplied.

The "Store" row also covers the store portion of an atomic RMW operation. In addition, the "Load" row covers the load component of a successful value-returning `_relaxed()` RMW atomic operation, although the combined "`_relaxed()` RMW operation" line provides a convenient combined reference in the value-returning case. A CPU executing unsuccessful value-returning atomic RMW operations must invalidate the corresponding variable from all other CPUs' caches. Therefore, unsuccessful value-returning atomic RMW operations have many of the properties of a store, which means that the "`_relaxed()` RMW operation" line also applies to unsuccessful value-returning atomic RMW operations.

The `*_acquire` row covers `smp_load_acquire()`, `cmpxchg_acquire()`, `xchg_acquire()`, and so on; the `*_release` row covers `smp_store_release()`, `rcu_assign_pointer()`, `cmpxchg_release()`, `xchg_release()`, and so on; and the "Successful full-strength non-`void` RMW" row covers `atomic_add_return()`, `atomic_add_unless()`, `atomic_dec_and_test()`, `cmpxchg()`, `xchg()`, and so on. The "Successful" qualifiers apply to primitives such as `atomic_add_unless()`, `cmpxchg_acquire()`, and `cmpxchg_release()`, which have no effect on either memory or on ordering when they indicate failure, as indicated by the earlier "`_relaxed()` RMW operation" row.

Column "C" indicates cumulativity and propagation, as explained in Sections 15.2.7.1 and 15.2.7.2. In the meantime, this column can usually be ignored when there are at most two threads involved.

Quick Quiz 15.5: The rows in Table 15.3 seem quite random and confused. Whatever is the conceptual basis of this table???
■

Table 15.2: Memory Ordering: Store-Buffering Sequence of Events

	CPU 0			CPU 1		
	Instruction	Store Buffer	Cache	Instruction	Store Buffer	Cache
1	(Initial state)		x1==0	(Initial state)		x0==0
2	x0 = 2;	x0==2	x1==0	x1 = 2;	x1==2	x0==0
3	smp_mb();	x0==2	x1==0	smp_mb();	x1==2	x0==0
4	(Read-invalidate)	x0==2	x0==0	(Read-invalidate)	x1==2	x1==0
5	(Finish store)		x0==2	(Finish store)		x1==2
6	r2 = x1; (2)		x1==2	r2 = x0; (2)		x0==2

Table 15.3: Linux-Kernel Memory-Ordering Cheat Sheet

		Prior Ordered Operation				Subsequent Ordered Operation						
Operation Providing Ordering	C	Self	R	W	RMW	Self	R	W	DR	DW	RMW	SV
Store, for example, WRITE_ONCE()		Y										Y
Load, for example, READ_ONCE()		Y							Y	Y		Y
_relaxed() RMW operation		Y							Y	Y		Y
*_dereference()		Y							Y	Y		Y
Successful *_acquire()		R					Y	Y	Y	Y	Y	Y
Successful *_release()	C		Y	Y	Y	W						Y
smp_rmb()			Y		R		Y		Y		R	
smp_wmb()				Y	W			Y		Y	W	
smp_mb() and synchronize_rcu()	CP		Y	Y	Y		Y	Y	Y	Y	Y	
Successful full-strength non-void RMW	CP	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
smp_mb__before_atomic()	CP		Y	Y	Y		a	a	a	a	Y	
smp_mb__after_atomic()	CP		a	a	Y		Y	Y	Y	Y	Y	

Key:
- C: Ordering is cumulative
- P: Ordering propagates
- R: Read, for example, READ_ONCE(), or read portion of RMW
- W: Write, for example, WRITE_ONCE(), or write portion of RMW
- Y: Provides the specified ordering
- a: Provides specified ordering given intervening RMW atomic operation
- DR: Dependent read (address dependency, Section 15.2.3)
- DW: Dependent write (address, data, or control dependency, Sections 15.2.3–15.2.5)
- RMW: Atomic read-modify-write operation
- Self: Orders self, as opposed to accesses both before and after
- SV: Orders later accesses to the same variable

Applies to Linux kernel v4.15 and later.

www.dbooks.org

Quick Quiz 15.6: Why is Table 15.3 missing `smp_mb__after_unlock_lock()` and `smp_mb__after_spinlock()`? ■

It is important to note that this table is just a cheat sheet, and is therefore in no way a replacement for a good understanding of memory ordering. To begin building such an understanding, the next section will present some basic rules of thumb.

15.1.3 Basic Rules of Thumb

This section presents some basic rules of thumb that are "good and sufficient" for a great many situations. In fact, you could write a great deal of concurrent code having excellent performance and scalability without needing anything more than these rules of thumb. More sophisticated rules of thumb will be presented in Section 15.6.

Quick Quiz 15.7: But how can I know that a given project can be designed and coded within the confines of these rules of thumb? ■

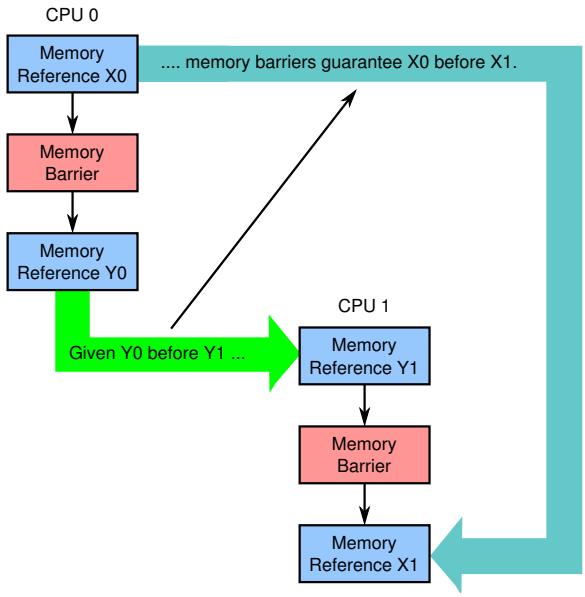


Figure 15.3: Memory Barriers Provide Conditional If-Then Ordering

A given thread sees its own accesses in order. This rule assumes that loads and stores from/to shared variables use `READ_ONCE()` and `WRITE_ONCE()`, respectively. Otherwise, the compiler can profoundly scramble[4] your code, and sometimes the CPU can do a bit of scrambling as well, as discussed in Section 15.5.4.

Ordering has conditional if-then semantics. Figure 15.3 illustrates this for memory barriers. Assuming that both memory barriers are strong enough, if CPU 1's access Y1 happens after CPU 0's access Y0, then CPU 1's access X1 is guaranteed to happen after CPU 0's access X1. When in doubt as to which memory barriers are strong enough, `smp_mb()` will always do the job, albeit at a price.

Quick Quiz 15.8: How can you tell which memory barriers are strong enough for a given use case? ■

Listing 15.2 is a case in point. The `smp_mb()` on lines 10 and 19 serve as the barriers, the store to `x0` on line 9 as X0, the load from `x1` on line 11 as Y0, the store to `x1` on line 18 as Y1, and the load from `x0` on line 20 as X1. Applying the if-then rule step by step, we know that the store to `x1` on line 18 happens after the load from `x1` on line 11 if `P0()`'s local variable `r2` is set to the value zero. The if-then rule would then state that the load from `x0` on line 20 happens after the store to `x0` on line 9. In other words, `P1()`'s local variable `r2` is guaranteed to end up with the value two *only if* `P0()`'s local variable `r2` ends up with the value zero. This underscores the point that memory ordering guarantees are conditional, not absolute.

Although Figure 15.3 specifically mentions memory barriers, this same if-then rule applies to the rest of the Linux kernel's ordering operations.

Ordering operations must be paired. If you carefully order the operations in one thread, but then fail to do so in another thread, then there is no ordering. Both threads must provide ordering for the if-then rule to apply.[5]

Ordering operations almost never speed things up. If you find yourself tempted to add a memory barrier in an attempt to force a prior store to be flushed to memory faster, resist! Adding ordering usually slows things down. Of course, there are situations where adding instructions speeds things up, as was shown by Figure 9.22 on page 157, but careful benchmarking is required in such cases. And even then, it is quite possible that although you sped things up a little bit on *your* system, you might well have slowed

[4] Many compiler writers prefer the word "optimize".

[5] In Section 15.2.7.2, pairing will be generalized to cycles.

things down significantly on your users' systems. Or on your future system.

Ordering operations are not magic. When your program is failing due to some race condition, it is often tempting to toss in a few memory-ordering operations in an attempt to barrier your bugs out of existence. A far better reaction is to use higher-level primitives in a carefully designed manner. With concurrent programming, it is almost always better to design your bugs out of existence than to hack them down to lower probabilities.

These are only rough rules of thumb. Although these rules of thumb cover the vast majority of situations seen in actual practice, as with any set of rules of thumb, they do have their limits. The next section will demonstrate some of these limits by introducing trick-and-trap litmus tests that are intended to insult your intuition while increasing your understanding. These litmus tests will also illuminate many of the concepts represented by the Linux-kernel memory-ordering cheat sheet shown in Table 15.3, and can be automatically analyzed given proper tooling [AMM+18]. Section 15.6 will circle back to this cheat sheet, presenting a more sophisticated set of rules of thumb in light of learnings from all the intervening tricks and traps.

Quick Quiz 15.9: Wait!!! Where do I find this tooling that automatically analyzes litmus tests??? ■

15.2 Tricks and Traps

> Knowing where the trap is—that's the first step in evading it.
>
> *Duke Leto Atreides, "Dune", Frank Herbert*

Now that you know that hardware can reorder memory accesses and that you can prevent it from doing so, the next step is to get you to admit that your intuition has a problem. This painful task is taken up by Section 15.2.1, which presents some code demonstrating that scalar variables can take on multiple values simultaneously, and by Sections 15.2.2 through 15.2.7, which show a series of intuitively correct code fragments that fail miserably on real hardware. Once your intuition has made it through the grieving process, later sections will summarize the basic rules that memory ordering follows.

Listing 15.3: Software Logic Analyzer

```
1 state.variable = mycpu;
2 lasttb = oldtb = firsttb = gettb();
3 while (state.variable == mycpu) {
4   lasttb = oldtb;
5   oldtb = gettb();
6   if (lasttb - firsttb > 1000)
7     break;
8 }
```

But first, let's take a quick look at just how many values a single variable might have at a single point in time.

15.2.1 Variables With Multiple Values

It is natural to think of a variable as taking on a well-defined sequence of values in a well-defined, global order. Unfortunately, the next stop on the journey says "goodbye" to this comforting fiction. Hopefully, you already started to say "goodbye" in response to row 2 of Tables 15.1 and 15.2, and if so, the purpose of this section is to drive this point home.

To this end, consider the program fragment shown in Listing 15.3. This code fragment is executed in parallel by several CPUs. Line 1 sets a shared variable to the current CPU's ID, line 2 initializes several variables from a `gettb()` function that delivers the value of a fine-grained hardware "timebase" counter that is synchronized among all CPUs (not available from all CPU architectures, unfortunately!), and the loop from lines 3–8 records the length of time that the variable retains the value that this CPU assigned to it. Of course, one of the CPUs will "win", and would thus never exit the loop if not for the check on lines 6–7.

Quick Quiz 15.10: What assumption is the code fragment in Listing 15.3 making that might not be valid on real hardware? ■

Upon exit from the loop, `firsttb` will hold a timestamp taken shortly after the assignment and `lasttb` will hold a timestamp taken before the last sampling of the shared variable that still retained the assigned value, or a value equal to `firsttb` if the shared variable had changed before entry into the loop. This allows us to plot each CPU's view of the value of `state.variable` over a 532-nanosecond time period, as shown in Figure 15.4. This data was collected in 2006 on 1.5 GHz POWER5 system with 8 cores, each containing a pair of hardware threads. CPUs 1, 2, 3, and 4 recorded the values, while CPU 0 controlled the test. The timebase counter period was about 5.32 ns, sufficiently fine-grained to allow observations of intermediate cache states.

www.dbooks.org

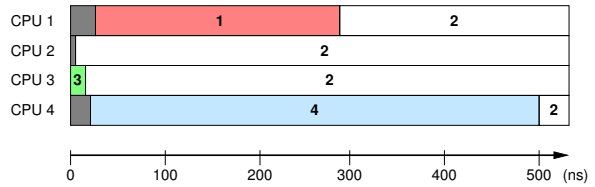


Figure 15.4: A Variable With Multiple Simultaneous Values

Each horizontal bar represents the observations of a given CPU over time, with the gray regions to the left indicating the time before the corresponding CPU's first measurement. During the first 5 ns, only CPU 3 has an opinion about the value of the variable. During the next 10 ns, CPUs 2 and 3 disagree on the value of the variable, but thereafter agree that the value is "2", which is in fact the final agreed-upon value. However, CPU 1 believes that the value is "1" for almost 300 ns, and CPU 4 believes that the value is "4" for almost 500 ns.

Quick Quiz 15.11: How could CPUs possibly have different views of the value of a single variable *at the same time?* ■

Quick Quiz 15.12: Why do CPUs 2 and 3 come to agreement so quickly, when it takes so long for CPUs 1 and 4 to come to the party? ■

And if you think that the situation with four CPUs was intriguing, consider Figure 15.5, which shows the same situation, but with 15 CPUs each assigning their number to a single shared variable at time $t = 0$. Both diagrams in the figure are drawn in the same way as Figure 15.4. The only difference is that the unit of horizontal axis is timebase ticks, with each tick lasting about 5.3 nanoseconds. The entire sequence therefore lasts a bit longer than the events recorded in Figure 15.4, consistent with the increase in number of CPUs. The upper diagram shows the overall picture, while the lower one zooms in on the first 50 timebase ticks. Again, CPU 0 coordinates the test, so does not record any values.

All CPUs eventually agree on the final value of 9, but not before the values 15 and 12 take early leads. Note that there are fourteen different opinions on the variable's value at time 21 indicated by the vertical line in the lower diagram. Note also that all CPUs see sequences whose orderings are consistent with the directed graph shown in Figure 15.6. Nevertheless, these figures underscore the importance of proper use of memory-ordering operations.

How many values can a single variable take on at a single point in time? As many as one per store buffer in the system! We have therefore entered a regime where we must bid a fond farewell to comfortable intuitions about values of variables and the passage of time. This is the regime where memory-ordering operations are needed.

But remember well the lessons from Chapters 3 and 6. Having all CPUs store concurrently to the same variable is no way to design a parallel program, at least not if performance and scalability are at all important to you.

Unfortunately, memory ordering has many other ways of insulting your intuition, and not all of these ways conflict with performance and scalability. The next section overviews reordering of unrelated memory reference.

Listing 15.4: Message-Passing Litmus Test (No Ordering)

```
1  C C-MP+o-wmb-o+o-o
2
3  {}
4
5  P0(int* x0, int* x1) {
6    WRITE_ONCE(*x0, 2);
7    smp_wmb();
8    WRITE_ONCE(*x1, 2);
9  }
10
11 P1(int* x0, int* x1) {
12   int r2;
13   int r3;
14
15   r2 = READ_ONCE(*x1);
16   r3 = READ_ONCE(*x0);
17 }
18
19 exists (1:r2=2 /\ 1:r3=0)
```

15.2.2 Memory-Reference Reordering

Section 15.1.1 showed that even relatively strongly ordered systems like x86 can reorder prior stores with later loads, at least when the store and load are to different variables. This section builds on that result, looking at the other combinations of loads and stores.

15.2.2.1 Load Followed By Load

Listing 15.4 (`C-MP+o-wmb-o+o-o.litmus`) shows the classic *message-passing* litmus test, where `x0` is the message and `x1` is a flag indicating whether or not a message is available. In this test, the `smp_wmb()` forces `P0()` stores to be ordered, but no ordering is specified for the loads. Relatively strongly ordered architectures, such as x86, do enforce ordering. However, weakly ordered architectures often do not [AMP+11]. Therefore, the `exists` clause on line 19 of the listing *can* trigger.

One rationale for reordering loads from different locations is that doing so allows execution to proceed when

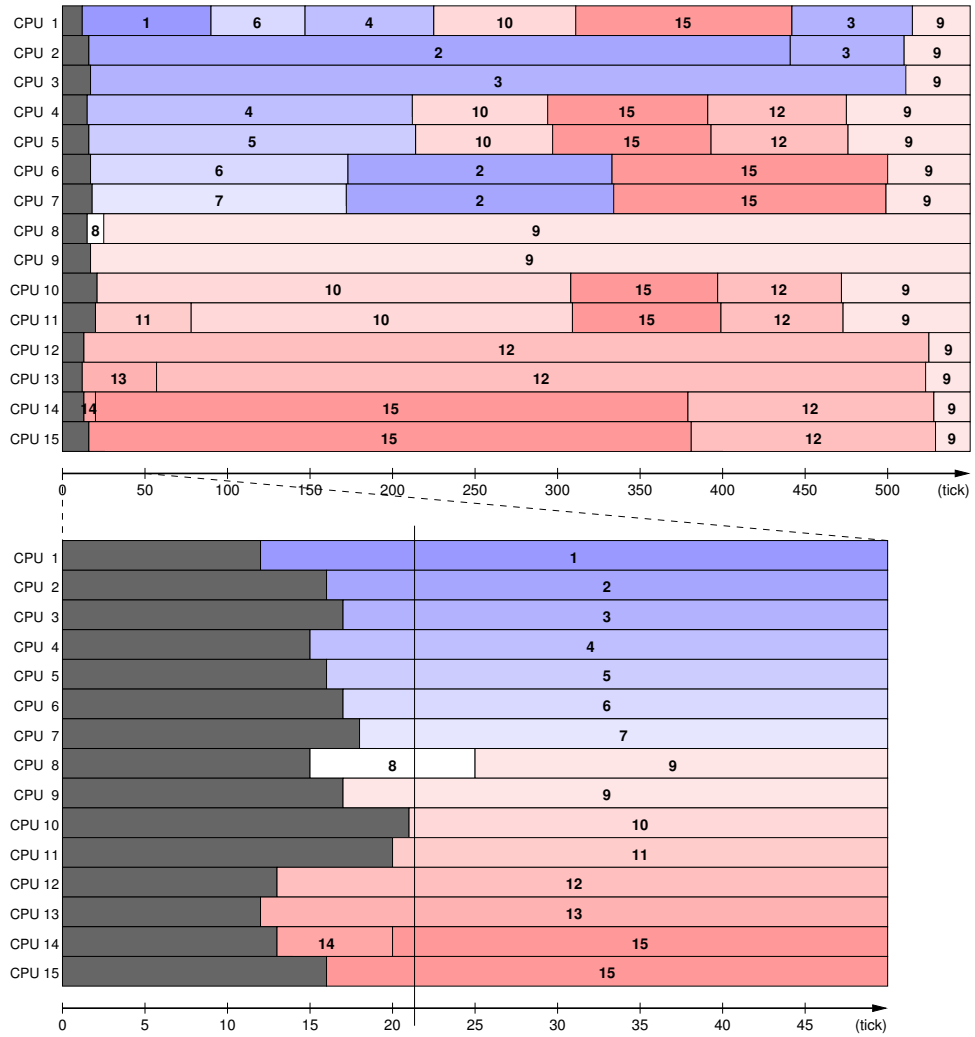


Figure 15.5: A Variable With More Simultaneous Values

www.dbooks.org

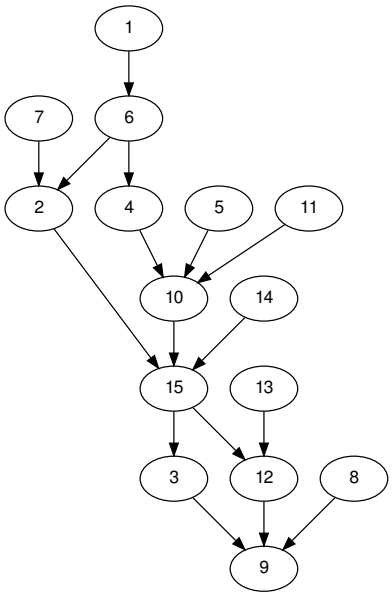


Figure 15.6: Possible Global Orders With More Simultaneous Values

an earlier load misses the cache, but the values for later loads are already present.

Quick Quiz 15.13: But why make load-load reordering visible to the user? Why not just use speculative execution to allow execution to proceed in the common case where there are no intervening stores, in which case the reordering cannot be visible anyway? ■

Thus, portable code relying on ordered loads must add explicit ordering, for example, the `smp_rmb()` shown on line 16 of Listing 15.5 (`C-MP+o-wmb-o+o-rmb-o.litmus`), which prevents the `exists_` clause from triggering.

Listing 15.5: Enforcing Order of Message-Passing Litmus Test

```
1  C C-MP+o-wmb-o+o-rmb-o
2
3  {}
4
5  P0(int* x0, int* x1) {
6    WRITE_ONCE(*x0, 2);
7    smp_wmb();
8    WRITE_ONCE(*x1, 2);
9  }
10
11 P1(int* x0, int* x1) {
12   int r2;
13   int r3;
14
15   r2 = READ_ONCE(*x1);
16   smp_rmb();
17   r3 = READ_ONCE(*x0);
18 }
19
20 exists (1:r2=2 /\ 1:r3=0)
```

Listing 15.6: Load-Buffering Litmus Test (No Ordering)

```
1  C C-LB+o-o+o-o
2
3  {}
4
5  P0(int *x0, int *x1)
6  {
7    int r2;
8
9    r2 = READ_ONCE(*x1);
10   WRITE_ONCE(*x0, 2);
11 }
12
13 P1(int *x0, int *x1)
14 {
15   int r2;
16
17   r2 = READ_ONCE(*x0);
18   WRITE_ONCE(*x1, 2);
19 }
20
21 exists (1:r2=2 /\ 0:r2=2)
```

Listing 15.7: Enforcing Ordering of Load-Buffering Litmus Test

```
1  C C-LB+o-r+a-o
2
3  {}
4
5  P0(int *x0, int *x1)
6  {
7    int r2;
8
9    r2 = READ_ONCE(*x1);
10   smp_store_release(x0, 2);
11 }
12
13 P1(int *x0, int *x1)
14 {
15   int r2;
16
17   r2 = smp_load_acquire(x0);
18   WRITE_ONCE(*x1, 2);
19 }
20
21 exists (1:r2=2 /\ 0:r2=2)
```

15.2.2.2 Load Followed By Store

Listing 15.6 (`C-LB+o-o+o-o.litmus`) shows the classic *load-buffering* litmus test. Although relatively strongly ordered systems such as x86 or the IBM Mainframe do not reorder prior loads with subsequent stores, many weakly ordered architectures really do allow such reordering [AMP+11]. Therefore, the `exists` clause on line 21 really can trigger.

Although it is rare for actual hardware to exhibit this reordering [Mar17], one situation where it might be desirable to do so is when a load misses the cache, the store buffer is nearly full, and the cacheline for a subsequent

Listing 15.8: Message-Passing Litmus Test, No Writer Ordering (No Ordering)

```
1 C C-MP+o-o+o-rmb-o
2
3 {}
4
5 P0(int* x0, int* x1) {
6   WRITE_ONCE(*x0, 2);
7   WRITE_ONCE(*x1, 2);
8 }
9
10 P1(int* x0, int* x1) {
11   int r2;
12   int r3;
13
14   r2 = READ_ONCE(*x1);
15   smp_rmb();
16   r3 = READ_ONCE(*x0);
17 }
18
19 exists (1:r2=2 /\ 1:r3=0)
```

store is ready at hand. Therefore, portable code must enforce any required ordering, for example, as shown in Listing 15.7 (`C-LB+o-r+a-o.litmus`). The `smp_store_release()` and `smp_load_acquire()` guarantee that the `exists` clause on line 21 never triggers.

15.2.2.3 Store Followed By Store

Listing 15.8 (`C-MP+o-o+o-rmb-o.litmus`) once again shows the classic message-passing litmus test, with the `smp_rmb()` providing ordering for `P1()`'s loads, but without any ordering for `P0()`'s stores. Again, the relatively strongly ordered architectures do enforce ordering, but weakly ordered architectures do not necessarily do so [AMP+11], which means that the `exists` clause can trigger. One situation in which such reordering could be beneficial is when the store buffer is full, another store is ready to execute, but the cacheline needed by the oldest store is not yet available. In this situation, allowing stores to complete out of order would allow execution to proceed. Therefore, portable code must explicitly order the stores, for example, as shown in Listing 15.5, thus preventing the `exists` clause from triggering.

Quick Quiz 15.14: Why should strongly ordered systems pay the performance price of unnecessary `smp_rmb()` and `smp_wmb()` invocations? Shouldn't weakly ordered systems shoulder the full cost of their misordering choices??? ■

15.2.3 Address Dependencies

An *address dependency* occurs when the value returned by a load instruction is used to compute the address used by a later memory-reference instruction.

Listing 15.9: Message-Passing Address-Dependency Litmus Test (No Ordering Before v4.15)

```
1 C C-MP+o-wmb-o+o-ad-o
2
3 {
4 y=1;
5 x1=y;
6 }
7
8 P0(int* x0, int** x1) {
9   WRITE_ONCE(*x0, 2);
10   smp_wmb();
11   WRITE_ONCE(*x1, x0);
12 }
13
14 P1(int** x1) {
15   int *r2;
16   int r3;
17
18   r2 = READ_ONCE(*x1);
19   r3 = READ_ONCE(*r2);
20 }
21
22 exists (1:r2=x0 /\ 1:r3=1)
```

Listing 15.9 (`C-MP+o-wmb-o+o-addr-o.litmus`) shows a linked variant of the message-passing pattern. The head pointer is `x1`, which initially references the `int` variable y (line 5), which is in turn initialized to the value 1 (line 4). `P0()` updates head pointer `x1` to reference `x0` (line 11), but only after initializing it to 2 (line 9) and forcing ordering (line 10). `P1()` picks up the head pointer `x1` (line 18), and then loads the referenced value (line 19). There is thus an address dependency from the load on line 18 to the load on line 19. In this case, the value returned by line 18 is exactly the address used by line 19, but many variations are possible, including field access using the C-language `->` operator, addition, subtraction, and array indexing.[6]

One might hope that line 18's load from the head pointer would be ordered before line 19's dereference, which is in fact the case on Linux v4.15 and later. However, prior to v4.15, this was not the case on DEC Alpha, which could in effect use a speculated value for the dependent load, as described in more detail in Section 15.5.1. Therefore, on older versions of Linux, Listing 15.9's `exists` clause *can* trigger.

Listing 15.10 shows how to make this work reliably on pre-v4.15 Linux kernels running on DEC Alpha, by replacing line 19's `READ_ONCE()` with `lockless_dereference()`,[7] which acts like `READ_ONCE()` on all

[6] But note that in the Linux kernel, the address dependency must be carried through the pointer to the array, not through the array index.

[7] Note that `lockless_dereference()` is not needed on v4.15 and later, and therefore is not available in these later Linux kernels. Nor is it needed in versions of this book containing this sentence.

www.dbooks.org

Listing 15.10: Enforced Ordering of Message-Passing Address-Dependency Litmus Test (Before v4.15)

```
1  C C-MP+o-wmb-o+ld-addr-o
2
3  {
4  y=1;
5  x1=y;
6  }
7
8  P0(int* x0, int** x1) {
9    WRITE_ONCE(*x0, 2);
10   smp_wmb();
11   WRITE_ONCE(*x1, x0);
12 }
13
14 P1(int** x1) {
15   int *r2;
16   int r3;
17
18   r2 = lockless_dereference(*x1); // Obsolete
19   r3 = READ_ONCE(*r2);
20 }
21
22 exists (1:r2=x0 /\ 1:r3=1)
```

Listing 15.11: S Address-Dependency Litmus Test

```
1  C C-S+o-wmb-o+o-addr-o
2
3  {
4  y=1;
5  x1=y;
6  }
7
8  P0(int* x0, int** x1) {
9    WRITE_ONCE(*x0, 2);
10   smp_wmb();
11   WRITE_ONCE(*x1, x0);
12 }
13
14 P1(int** x1) {
15   int *r2;
16
17   r2 = READ_ONCE(*x1);
18   WRITE_ONCE(*r2, 3);
19 }
20
21 exists (1:r2=x0 /\ x0=2)
```

platforms other than DEC Alpha, where it acts like a `READ_ONCE()` followed by an `smp_mb()`, thereby forcing the required ordering on all platforms, in turn preventing the `exists` clause from triggering.

But what happens if the dependent operation is a store rather than a load, for example, in the *S* litmus test [AMP+11] shown in Listing 15.11 (`C-S+o-wmb-o+o-addr-o.litmus`)? Because no production-quality platform speculates stores, it is not possible for the `WRITE_ONCE()` on line 9 to overwrite the `WRITE_ONCE()` on line 18, meaning that the `exists` clause on line 21 cannot trigger, even on DEC Alpha, even in pre-v4.15 Linux kernels.

Listing 15.12: Load-Buffering Data-Dependency Litmus Test

```
1  C C-LB+o-r+o-data-o
2
3  {}
4
5  P0(int *x0, int *x1)
6  {
7    int r2;
8
9    r2 = READ_ONCE(*x1);
10   smp_store_release(x0, 2);
11 }
12
13 P1(int *x0, int *x1)
14 {
15   int r2;
16
17   r2 = READ_ONCE(*x0);
18   WRITE_ONCE(*x1, r2);
19 }
20
21 exists (1:r2=2 /\ 0:r2=2)
```

Quick Quiz 15.15: But how do we know that *all* platforms really avoid triggering the `exists` clauses in Listings 15.10 and 15.11? ■

Quick Quiz 15.16: SP, MP, LB, and now S. Where do all these litmus-test abbreviations come from and how can anyone keep track of them? ■

However, it is important to note that address dependencies can be fragile and easily broken by compiler optimizations, as discussed in Section 15.3.2.

15.2.4 Data Dependencies

A *data dependency* occurs when the value returned by a load instruction is used to compute the data stored by a later store instruction. Note well the "data" above: If the value returned by a load was instead used to compute the address used by a later store instruction, that would instead be an address dependency.

Listing 15.12 (`C-LB+o-r+o-data-o.litmus`) is similar to Listing 15.7, except that `P1()`'s ordering between lines 17 and 18 is enforced not by an acquire load, but instead by a data dependency: The value loaded by line 17 is what line 18 stores. The ordering provided by this data dependency is sufficient to prevent the `exists` clause from triggering.

Just as with address dependencies, data dependencies are fragile and can be easily broken by compiler optimizations, as discussed in Section 15.3.2. In fact, data dependencies can be even more fragile than are address dependencies. The reason for this is that address dependencies normally involve pointer values. In contrast, as

Listing 15.13: Load-Buffering Control-Dependency Litmus Test

```
1  C C-LB+o-r+o-ctrl-o
2
3  {}
4
5  P0(int *x0, int *x1)
6  {
7    int r2;
8
9    r2 = READ_ONCE(*x1);
10   smp_store_release(x0, 2);
11 }
12
13 P1(int *x0, int *x1)
14 {
15   int r2;
16
17   r2 = READ_ONCE(*x0);
18   if (r2 >= 0)
19     WRITE_ONCE(*x1, 2);
20 }
21
22 exists (1:r2=2 /\ 0:r2=2)
```

Listing 15.14: Message-Passing Control-Dependency Litmus Test (No Ordering)

```
1  C C-MP+o-r+o-ctrl-o
2
3  {}
4
5  P0(int* x0, int* x1) {
6    WRITE_ONCE(*x0, 2);
7    smp_store_release(x1, 2);
8  }
9
10 P1(int* x0, int* x1) {
11   int r2;
12   int r3 = 0;
13
14   r2 = READ_ONCE(*x1);
15   if (r2 >= 0)
16     r3 = READ_ONCE(*x0);
17 }
18
19 exists (1:r2=2 /\ 1:r3=0)
```

shown in Listing 15.12, it is tempting to carry data dependencies through integral values, which the compiler has much more freedom to optimize into nonexistence. For but one example, if the integer loaded was multiplied by the constant zero, the compiler would know that the result was zero, and could therefore substitute the constant zero for the value loaded, thus breaking the dependency.

Quick Quiz 15.17: But wait!!! Line 17 of Listing 15.12 uses `READ_ONCE()`, which marks the load as volatile, which means that the compiler absolutely must emit the load instruction even if the value is later multiplied by zero. So how can the compiler possibly break this data dependency? ■

In short, you can rely on data dependencies only if you prevent the compiler from breaking them.

15.2.5 Control Dependencies

A *control dependency* occurs when the value returned by a load instruction is tested to determine whether or not a later store instruction is executed. Note well the "later store instruction": Many platforms do not respect load-to-load control dependencies.

Listing 15.13 (`C-LB+o-r+o-ctrl-o.litmus`) shows another load-buffering example, this time using a control dependency (line 18) to order the load on line 17 and the store on line 19. The ordering is sufficient to prevent the `exists` from triggering.

However, control dependencies are even more susceptible to being optimized out of existence than are data dependencies, and Section 15.3.3 describes some of the rules that must be followed in order to prevent your compiler from breaking your control dependencies.

It is worth reiterating that control dependencies provide ordering only from loads to stores. Therefore, the load-to-load control dependency shown on lines 14–16 of Listing 15.14 (`C-MP+o-r+o-ctrl-o.litmus`) does *not* provide ordering, and therefore does *not* prevent the `exists` clause from triggering.

In summary, control dependencies can be useful, but they are high-maintenance items. You should therefore use them only when performance considerations permit no other solution.

Quick Quiz 15.18: Wouldn't control dependencies be more robust if they were mandated by language standards??? ■

15.2.6 Cache Coherence

On cache-coherent platforms, all CPUs agree on the order of loads and stores to a given variable. Fortunately, when `READ_ONCE()` and `WRITE_ONCE()` are used, almost all platforms are cache-coherent, as indicated by the "SV" column of the cheat sheet shown in Table 15.3. Unfortunately, this property is so popular that it has been named multiple times, with "single-variable SC",[8] "single-copy atomic" [SF95], and just plain "coherence" [AMP+11] having seen use. Rather than further compound the confusion by inventing yet another term for this concept, this book uses "cache coherence" and "coherence" interchangeably.

Listing 15.15 (`C-CCIRIW+o+o+o-o+o-o.litmus`) shows a litmus test that tests for cache coherence, where

[8] Recall that SC stands for sequentially consistent.

www.dbooks.org

Listing 15.15: Cache-Coherent IRIW Litmus Test

```
1  C C-CCIRIW+o+o+o-o+o-o
2
3  {}
4
5  P0(int *x)
6  {
7    WRITE_ONCE(*x, 1);
8  }
9
10 P1(int *x)
11 {
12   WRITE_ONCE(*x, 2);
13 }
14
15 P2(int *x)
16 {
17   int r1;
18   int r2;
19
20   r1 = READ_ONCE(*x);
21   r2 = READ_ONCE(*x);
22 }
23
24 P3(int *x)
25 {
26   int r3;
27   int r4;
28
29   r3 = READ_ONCE(*x);
30   r4 = READ_ONCE(*x);
31 }
32
33 exists(2:r1=1 /\ 2:r2=2 /\ 3:r3=2 /\ 3:r4=1)
```

"IRIW" stands for "independent reads of independent writes". Because this litmus test uses only one variable, `P2()` and `P3()` must agree on the order of `P0()`'s and `P1()`'s stores. In other words, if `P2()` believes that `P0()`'s store came first, then `P3()` had better not believe that `P1()`'s store came first. And in fact the `exists` clause on line 33 will trigger if this situation arises.

Quick Quiz 15.19: But in Listing 15.15, wouldn't be just as bad if `P2()`'s `r1` and `r2` obtained the values 2 and 1, respectively, while `P3()`'s `r3` and `r4` obtained the values 1 and 2, respectively? ■

It is tempting to speculate that different-sized overlapping loads and stores to a single region of memory (as might be set up using the C-language `union` keyword) would provide similar ordering guarantees. However, Flur et al. [FSP+17] discovered some surprisingly simple litmus tests that demonstrate that such guarantees can be violated on real hardware. It is therefore necessary to restrict code to non-overlapping same-sized aligned accesses to a given variable, at least if portability is a consideration.[9]

[9] There is reason to believe that using atomic RMW operations (for example, `xchg()`) for all the stores will provide sequentially consistent ordering, but this has not yet been proven either way.

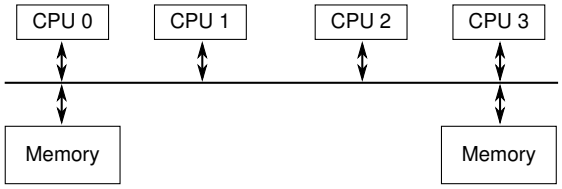


Figure 15.7: Global System Bus And Multi-Copy Atomicity

Adding more variables and threads increases the scope for reordering and other counter-intuitive behavior, as discussed in the next section.

15.2.7 Multicopy Atomicity

Threads running on a fully *multicopy atomic* [SF95] platform are guaranteed to agree on the order of stores, even to different variables. A useful mental model of such a system is the single-bus architecture shown in Figure 15.7. If each store resulted in a message on the bus, and if the bus could accommodate only one store at a time, then any pair of CPUs would agree on the order of all stores that they observed. Unfortunately, building a computer system as shown in the figure, without store buffers or even caches, would result in glacially slow computation. Most CPU vendors interested in providing multicopy atomicity therefore instead provid the slightly weaker *other-multicopy atomicity* [ARM17, Section B2.3], which excludes the CPU doing a given store from the requirement that all CPUs agree on the order of all stores.[10] This means that if only a subset of CPUs are doing stores, the other CPUs will agree on the order of stores, hence the "other" in "other-multicopy atomicity". Unlike multicopy-atomic platforms, within other-multicopy-atomic platforms, the CPU doing the store is permitted to observe its store early, which allows its later loads to obtain the newly stored value directly from the store buffer, which improves performance.

Quick Quiz 15.20: Can you give a specific example showing different behavior for multicopy atomic on the one hand and other-multicopy atomic on the other? ■

Perhaps there will come a day when all platforms provide some flavor of multi-copy atomicity, but in the

[10] As of early 2021, Armv8 and x86 provide other-multicopy atomicity, IBM mainframe provides full multicopy atomicity, and PPC provides no multicopy atomicity at all. More detail is shown in Table 15.5.

Listing 15.16: WRC Litmus Test With Dependencies (No Ordering)

```
1  C C-WRC+o+o-data-o+o-rmb-o
2
3  {}
4
5  P0(int *x)
6  {
7    WRITE_ONCE(*x, 1);
8  }
9
10 P1(int *x, int* y)
11 {
12   int r1;
13
14   r1 = READ_ONCE(*x);
15   WRITE_ONCE(*y, r1);
16 }
17
18 P2(int *x, int* y)
19 {
20   int r2;
21   int r3;
22
23   r2 = READ_ONCE(*y);
24   smp_rmb();
25   r3 = READ_ONCE(*x);
26 }
27
28 exists (1:r1=1 /\ 2:r2=1 /\ 2:r3=0)
```

meantime, non-multicopy-atomic platforms do exist, and so software must deal with them.

Listing 15.16 (`C-WRC+o+o-data-o+o-rmb-o.litmus`) demonstrates multicopy atomicity, that is, on a multicopy-atomic platform, the `exists` clause on line 28 cannot trigger. In contrast, on a non-multicopy-atomic platform this `exists` clause can trigger, despite `P1()`'s accesses being ordered by a data dependency and `P2()`'s accesses being ordered by an `smp_rmb()`. Recall that the definition of multicopy atomicity requires that all threads agree on the order of stores, which can be thought of as all stores reaching all threads at the same time. Therefore, a non-multicopy-atomic platform can have a store reach different threads at different times. In particular, `P0()`'s store might reach `P1()` long before it reaches `P2()`, which raises the possibility that `P1()`'s store might reach `P2()` before `P0()`'s store does.

This leads to the question of why a real system constrained by the usual laws of physics would ever trigger the `exists` clause of Listing 15.16. The cartoonish diagram of a such a real system is shown in Figure 15.8. CPU 0 and CPU 1 share a store buffer, as do CPUs 2 and 3. This means that CPU 1 can load a value out of the store buffer, thus potentially immediately seeing a value stored by CPU 0. In contrast, CPUs 2 and 3 will have to wait for the corresponding cache line to carry this new value to them.

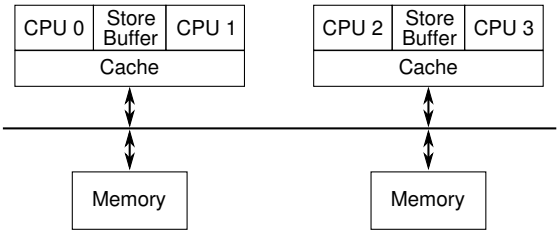


Figure 15.8: Shared Store Buffers And Multi-Copy Atomicity

Quick Quiz 15.21: Then who would even *think* of designing a system with shared store buffers??? ■

Table 15.4 shows one sequence of events that can result in the `exists` clause in Listing 15.16 triggering. This sequence of events will depend critically on `P0()` and `P1()` sharing both cache and a store buffer in the manner shown in Figure 15.8.

Quick Quiz 15.22: But just how is it fair that `P0()` and `P1()` must share a store buffer and a cache, but `P2()` gets one each of its very own??? ■

Row 1 shows the initial state, with the initial value of `y` in `P0()`'s and `P1()`'s shared cache, and the initial value of `x` in `P2()`'s cache.

Row 2 shows the immediate effect of `P0()` executing its store on line 7. Because the cacheline containing `x` is not in `P0()`'s and `P1()`'s shared cache, the new value (`1`) is stored in the shared store buffer.

Row 3 shows two transitions. First, `P0()` issues a read-invalidate operation to fetch the cacheline containing `x` so that it can flush the new value for `x` out of the shared store buffer. Second, `P1()` loads from `x` (line 14), an operation that completes immediately because the new value of `x` is immediately available from the shared store buffer.

Row 4 also shows two transitions. First, it shows the immediate effect of `P1()` executing its store to `y` (line 15), placing the new value into the shared store buffer. Second, it shows the start of `P2()`'s load from `y` (line 23).

Row 5 continues the tradition of showing two transitions. First, it shows `P1()` complete its store to `y`, flushing from the shared store buffer to the cache. Second, it shows `P2()` request the cacheline containing `y`.

Row 6 shows `P2()` receive the cacheline containing `y`, allowing it to finish its load into `r2`, which takes on the value `1`.

Row 7 shows `P2()` execute its `smp_rmb()` (line 24), thus keeping its two loads ordered.

www.dbooks.org

Table 15.4: Memory Ordering: WRC Sequence of Events

	P0()	P0() & P1()		P1()	P2()		
	Instruction	Store Buffer	Cache	Instruction	Instruction	Store Buffer	Cache
1	(Initial state)		y==0	(Initial state)	(Initial state)		x==0
2	x = 1;	x==1	y==0				x==0
3	(Read-Invalidate x)	x==1	y==0	r1 = x (1)			x==0
4		x==1 y==1	y==0	y = r1	r2 = y		x==0
5		x==1	y==1	(Finish store)	(Read y)		x==0
6	(Respond y)	x==1	y==1		(r2==1)		x==0 y==1
7		x==1	y==1		smp_rmb()		x==0 y==1
8		x==1	y==1		r3 = x (0)		x==0 y==1
9		x==1	x==0 y==1		(Respond x)		y==1
10	(Finish store)		x==1 y==1				y==1

Row 8 shows P2() execute its load from x, which immediately returns with the value zero from P2()'s cache.

Row 9 shows P2() *finally* responding to P0()'s request for the cacheline containing x, which was made way back up on row 3.

Finally, row 10 shows P0() finish its store, flushing its value of x from the shared store buffer to the shared cache.

Note well that the exists clause on line 28 has triggered. The values of r1 and r2 are both the value one, and the final value of r3 the value zero. This strange result occurred because P0()'s new value of x was communicated to P1() long before it was communicated to P2().

Quick Quiz 15.23: Referring to Table 15.4, why on earth would P0()'s store take so long to complete when P1()'s store complete so quickly? In other words, does the exists clause on line 28 of Listing 15.16 really trigger on real systems? ■

This counter-intuitive result happens because although dependencies do provide ordering, they provide it only within the confines of their own thread. This three-thread example requires stronger ordering, which is the subject of Sections 15.2.7.1 through 15.2.7.4.

15.2.7.1 Cumulativity

The three-thread example shown in Listing 15.16 requires *cumulative* ordering, or *cumulativity*. A cumulative memory-ordering operation orders not just any given access preceding it, but also earlier accesses by any thread to that same variable.

Dependencies do not provide cumulativity, which is why the "C" column is blank for the READ_ONCE() row of Table 15.3 on page 301. However, as indicated by the "C" in their "C" column, release operations do provide cumulativity. Therefore, Listing 15.17 (C-WRC+o+o-r+a-o.litmus) substitutes a release operation for Listing 15.16's data dependency. Because the release operation is cumulative, its ordering applies not only to Listing 15.17's load from x by P1() on line 14, but also to the store to x by P0() on line 7—but only if that load returns the value stored, which matches the 1:r1=1 in the exists clause on line 27. This means that P2()'s load-acquire suffices to force the load from x on line 24 to happen after the store on line 7, so the value

Listing 15.17: WRC Litmus Test With Release

```
1  C C-WRC+o+o-r+a-o
2
3  {}
4
5  P0(int *x)
6  {
7    WRITE_ONCE(*x, 1);
8  }
9
10 P1(int *x, int* y)
11 {
12   int r1;
13
14   r1 = READ_ONCE(*x);
15   smp_store_release(y, r1);
16 }
17
18 P2(int *x, int* y)
19 {
20   int r2;
21   int r3;
22
23   r2 = smp_load_acquire(y);
24   r3 = READ_ONCE(*x);
25 }
26
27 exists (1:r1=1 /\ 2:r2=1 /\ 2:r3=0)
```

Listing 15.18: W+RWC Litmus Test With Release (No Ordering)

```
1  C C-W+RWC+o-r+a-o+o-mb-o
2
3  {}
4
5  P0(int *x, int *y)
6  {
7    WRITE_ONCE(*x, 1);
8    smp_store_release(y, 1);
9  }
10
11 P1(int *y, int *z)
12 {
13   int r1;
14   int r2;
15
16   r1 = smp_load_acquire(y);
17   r2 = READ_ONCE(*z);
18 }
19
20 P2(int *z, int *x)
21 {
22   int r3;
23
24   WRITE_ONCE(*z, 1);
25   smp_mb();
26   r3 = READ_ONCE(*x);
27 }
28
29 exists(1:r1=1 /\ 1:r2=0 /\ 2:r3=0)
```

returned is one, which does not match `2:r3=0`, which in turn prevents the `exists` clause from triggering.

These ordering constraints are depicted graphically in Figure 15.9. Note also that cumulativity is not limited to a single step back in time. If there was another load from `x` or store to `x` from any thread that came before the store on line 7, that prior load or store would also be ordered before the load on line 24, though only if both `r1` and `r2` both end up containing the value `1`.

In short, use of cumulative ordering operations can suppress non-multicopy-atomic behaviors in some situations. Cumulativity nevertheless has limits, which are examined in the next section.

15.2.7.2 Propagation

Listing 15.18 (`C-W+RWC+o-r+a-o+o-mb-o.litmus`) shows the limitations of cumulativity and store-release, even with a full memory barrier. The problem is that although the `smp_store_release()` on line 8 has cumulativity, and although that cumulativity does order `P2()`'s load on line 26, the `smp_store_release()`'s ordering cannot propagate through the combination of `P1()`'s load (line 17) and `P2()`'s store (line 24). This means that the `exists` clause on line 29 really can trigger.

Quick Quiz 15.24: But it is not necessary to worry about propagation unless there are at least three threads in the litmus test, right? ■

This situation might seem completely counter-intuitive, but keep in mind that the speed of light is finite and computers are of non-zero size. It therefore takes time for the effect of the `P2()`'s store to `z` to propagate to `P1()`, which in turn means that it is possible that `P1()`'s read from `z` happens much later in time, but nevertheless still sees the old value of zero. This situation is depicted in Figure 15.10: Just because a load sees the old value does *not* mean that this load executed at an earlier time than did the store of the new value.

Note that Listing 15.18 also shows the limitations of memory-barrier pairing, given that there are not two but three processes. These more complex litmus tests can instead be said to have *cycles*, where memory-barrier pairing is the special case of a two-thread cycle. The cycle in Listing 15.18 goes through `P0()` (lines 7 and 8), `P1()` (lines 16 and 17), `P2()` (lines 24, 25, and 26), and back to `P0()` (line 7). The `exists` clause delineates this cycle: the `1:r1=1` indicates that the `smp_load_acquire()` on line 16 returned the value stored by the `smp_store_release()` on line 8, the `1:r2=0` indicates that the `WRITE_ONCE()` on line 24 came too late to affect the value returned by the `READ_ONCE()` on line 17, and finally the `2:r3=0` indicates that the `WRITE_ONCE()` on line 7 came too late to affect the value returned by the `READ_ONCE()` on line 26. In this case, the fact that the `exists` clause can trigger means that the cycle is said to be *allowed*. In contrast, in cases where the `exists` clause cannot trigger, the cycle is said to be *prohibited*.

But what if we need to keep the `exists` clause on line 29 of Listing 15.18? One solution is to replace `P0()`'s `smp_store_release()` with an `smp_mb()`, which Table 15.3 shows to have not only cumulativity, but also propagation. The result is shown in Listing 15.19 (`C-W+RWC+o-mb-o+a-o+o-mb-o.litmus`).

Quick Quiz 15.25: But given that `smp_mb()` has the propagation property, why doesn't the `smp_mb()` on line 25 of Listing 15.18 prevent the `exists` clause from triggering? ■

For completeness, Figure 15.11 shows that the "winning" store among a group of stores to the same variable is not necessarily the store that started last. This should not come as a surprise to anyone who carefully examined Figure 15.5 on page 305.

www.dbooks.org

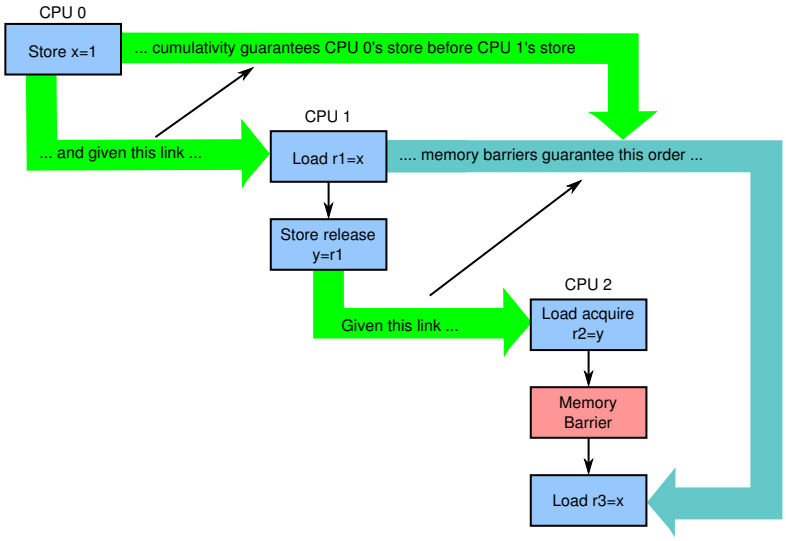


Figure 15.9: Cumulativity

Listing 15.19: W+WRC Litmus Test With More Barriers

```
1  C C-W+RWC+o-mb-o+a-o+o-mb-o
2
3  {}
4
5  P0(int *x, int *y)
6  {
7    WRITE_ONCE(*x, 1);
8    smp_mb();
9    WRITE_ONCE(*y, 1);
10 }
11
12 P1(int *y, int *z)
13 {
14   int r1;
15   int r2;
16
17   r1 = smp_load_acquire(y);
18   r2 = READ_ONCE(*z);
19 }
20
21 P2(int *z, int *x)
22 {
23   int r3;
24
25   WRITE_ONCE(*z, 1);
26   smp_mb();
27   r3 = READ_ONCE(*x);
28 }
29
30 exists(1:r1=1 /\ 1:r2=0 /\ 2:r3=0)
```

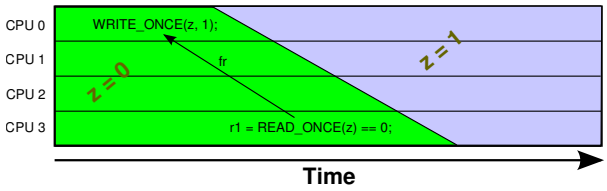


Figure 15.10: Load-to-Store is Counter-Temporal

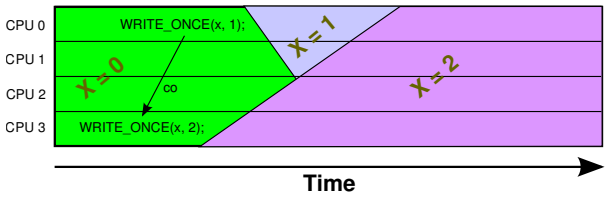


Figure 15.11: Store-to-Store is Counter-Temporal

Quick Quiz 15.26: But for litmus tests having only ordered stores, as shown in Listing 15.20 (`C-2+2W+o-wmb-o+o-wmb-o.litmus`), research shows that the cycle is prohibited, even in weakly ordered systems such as Arm and Power [SSA+11]. Given that, are store-to-store really *always* counter-temporal???
■

But sometimes time really is on our side. Read on!

Listing 15.20: 2+2W Litmus Test With Write Barriers

```
1  C C-2+2W+o-wmb-o+o-wmb-o
2
3  {}
4
5  P0(int *x0, int *x1)
6  {
7    WRITE_ONCE(*x0, 1);
8    smp_wmb();
9    WRITE_ONCE(*x1, 2);
10 }
11
12 P1(int *x0, int *x1)
13 {
14   WRITE_ONCE(*x1, 1);
15   smp_wmb();
16   WRITE_ONCE(*x0, 2);
17 }
18
19 exists (x0=1 /\ x1=1)
```

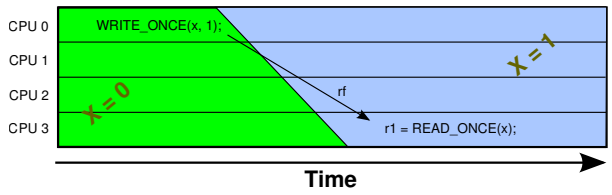


Figure 15.12: Store-to-Load is Temporal

15.2.7.3 Happens-Before

As shown in Figure 15.12, on platforms without user-visible speculation, if a load returns the value from a particular store, then, courtesy of the finite speed of light and the non-zero size of modern computing systems, the store absolutely has to have executed at an earlier time than did the load. This means that carefully constructed programs can rely on the passage of time itself as an memory-ordering operation.

Of course, just the passage of time by itself is not enough, as was seen in Listing 15.6 on page 306, which has nothing but store-to-load links and, because it provides absolutely no ordering, still can trigger its `exists` clause. However, as long as each thread provides even the weakest possible ordering, `exists` clause would not be able to trigger. For example, Listing 15.21 (`C-LB+a-o+o-data-o+o-data-o.litmus`) shows `P0()` ordered with an `smp_load_acquire()` and both `P1()` and `P2()` ordered with data dependencies. These orderings, which are close to the top of Table 15.3, suffice to prevent the `exists` clause from triggering.

Quick Quiz 15.27: Can you construct a litmus test like that in Listing 15.21 that uses *only* dependencies? ■

Listing 15.21: LB Litmus Test With One Acquire

```
1  C C-LB+a-o+o-data-o+o-data-o
2
3  {}
4
5  P0(int *x0, int *x1)
6  {
7    int r2;
8
9    r2 = smp_load_acquire(x0);
10   WRITE_ONCE(*x1, 2);
11 }
12
13 P1(int *x1, int *x2)
14 {
15   int r2;
16
17   r2 = READ_ONCE(*x1);
18   WRITE_ONCE(*x2, r2);
19 }
20
21 P2(int *x2, int *x0)
22 {
23   int r2;
24
25   r2 = READ_ONCE(*x2);
26   WRITE_ONCE(*x0, r2);
27 }
28
29 exists (0:r2=2 /\ 1:r2=2 /\ 2:r2=2)
```

An important use of time for ordering memory accesses is covered in the next section.

15.2.7.4 Release-Acquire Chains

A minimal release-acquire chain was shown in Listing 15.7 on page 306, but these chains can be much longer, as shown in Listing 15.22 (`C-LB+a-r+a-r+a-r+a-r+a-r.litmus`). The longer the release-acquire chain, the more ordering is gained from the passage of time, so that no matter how many threads are involved, the corresponding `exists` clause cannot trigger.

Although release-acquire chains are inherently store-to-load creatures, it turns out that they can tolerate one load-to-store step, despite such steps being counter-temporal, as shown in Figure 15.10 on page 314. For example, Listing 15.23 (`C-ISA2+o-r+a-r+a-r+a-o.litmus`) shows a three-step release-acquire chain, but where `P3()`'s final access is a `READ_ONCE()` from `x0`, which is accessed via `WRITE_ONCE()` by `P0()`, forming a non-temporal load-to-store link between these two processes. However, because `P0()`'s `smp_store_release()` (line 8) is cumulative, if `P3()`'s `READ_ONCE()` returns zero, this cumulativity will force the `READ_ONCE()` to be ordered before `P0()`'s `smp_store_release()`. In addition, the release-acquire chain (lines 8, 15, 16, 23, 24, and 32) forces `P3()`'s `READ_ONCE()` to be ordered after `P0()`'s `smp_store_`

www.dbooks.org

Listing 15.22: Long LB Release-Acquire Chain

```
1 C C-LB+a-r+a-r+a-r+a-r
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7   int r2;
8
9   r2 = smp_load_acquire(x0);
10  smp_store_release(x1, 2);
11 }
12
13 P1(int *x1, int *x2)
14 {
15  int r2;
16
17  r2 = smp_load_acquire(x1);
18  smp_store_release(x2, 2);
19 }
20
21 P2(int *x2, int *x3)
22 {
23  int r2;
24
25  r2 = smp_load_acquire(x2);
26  smp_store_release(x3, 2);
27 }
28
29 P3(int *x3, int *x0)
30 {
31  int r2;
32
33  r2 = smp_load_acquire(x3);
34  smp_store_release(x0, 2);
35 }
36
37 exists (0:r2=2 /\ 1:r2=2 /\ 2:r2=2 /\ 3:r2=2)
```

Listing 15.23: Long ISA2 Release-Acquire Chain

```
1 C C-ISA2+o-r+a-r+a-r+a-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7   WRITE_ONCE(*x0, 2);
8   smp_store_release(x1, 2);
9 }
10
11 P1(int *x1, int *x2)
12 {
13  int r2;
14
15  r2 = smp_load_acquire(x1);
16  smp_store_release(x2, 2);
17 }
18
19 P2(int *x2, int *x3)
20 {
21  int r2;
22
23  r2 = smp_load_acquire(x2);
24  smp_store_release(x3, 2);
25 }
26
27 P3(int *x3, int *x0)
28 {
29  int r1;
30  int r2;
31
32  r1 = smp_load_acquire(x3);
33  r2 = READ_ONCE(*x0);
34 }
35
36 exists (1:r2=2 /\ 2:r2=2 /\ 3:r1=2 /\ 3:r2=0)
```

`release()`. Because `P3()`'s `READ_ONCE()` cannot be both before and after `P0()`'s `smp_store_release()`, either or both of two things must be true:

1. `P3()`'s `READ_ONCE()` came after `P0()`'s `WRITE_ONCE()`, so that the `READ_ONCE()` returned the value two, so that the `exists` clause's `3:r2=0` is false.

2. The release-acquire chain did not form, that is, one or more of the `exists` clause's `1:r2=2`, `2:r2=2`, or `3:r1=2` is false.

Either way, the `exists` clause cannot trigger, despite this litmus test containing a notorious load-to-store link between `P3()` and `P0()`. But never forget that release-acquire chains can tolerate only one load-to-store link, as was seen in Listing 15.18.

Release-acquire chains can also tolerate a single store-to-store step, as shown in Listing 15.24 (`C-Z6.2+o-r+a-r+a-r+a-o.litmus`). As with the previous example, `smp_store_release()`'s cumulativity combined with the temporal nature of the release-acquire chain prevents the `exists` clause on line 35 from triggering. But beware: Adding a second store-to-store step would allow the correspondingly updated `exists` clause to trigger.

Quick Quiz 15.28: Suppose we have a short release-acquire chain along with one load-to-store link and one store-to-store link, like that shown in Listing 15.25. Given that there is only one of each type of non-store-to-load link, the `exists` cannot trigger, right? ■

Quick Quiz 15.29: There are store-to-load links, load-to-store links, and store-to-store links. But what about load-to-load links? ■

In short, properly constructed release-acquire chains form a peaceful island of intuitive bliss surrounded by a strongly counter-intuitive sea of more complex memory-ordering constraints.

Listing 15.24: Long Z6.2 Release-Acquire Chain

```
C C-Z6.2+o-r+a-r+a-r+a-o

{}

P0(int *x0, int *x1)
{
  WRITE_ONCE(*x0, 2);
  smp_store_release(x1, 2);
}

P1(int *x1, int *x2)
{
  int r2;

  r2 = smp_load_acquire(x1);
  smp_store_release(x2, 2);
}

P2(int *x2, int *x3)
{
  int r2;

  r2 = smp_load_acquire(x2);
  smp_store_release(x3, 2);
}

P3(int *x3, int *x0)
{
  int r2;

  r2 = smp_load_acquire(x3);
  WRITE_ONCE(*x0, 3);
}

exists (1:r2=2 /\ 2:r2=2 /\ 3:r2=2 /\ x0=2)
```

Listing 15.25: Z6.0 Release-Acquire Chain (Ordering?)

```
C C-Z6.2+o-r+a-o+o-mb-o

{}

P0(int *x, int *y)
{
  WRITE_ONCE(*x, 1);
  smp_store_release(y, 1);
}

P1(int *y, int *z)
{
  int r1;

  r1 = smp_load_acquire(y);
  WRITE_ONCE(*z, 1);
}

P2(int *z, int *x)
{
  int r2;

  WRITE_ONCE(*z, 2);
  smp_mb();
  r2 = READ_ONCE(*x);
}

exists(1:r1=1 /\ 2:r2=0 /\ z=2)
```

15.3 Compile-Time Consternation

> Science increases our power in proportion as it lowers our pride.
>
> *Claude Bernard*

Most languages, including C, were developed on uniprocessor systems by people with little or no parallel-programming experience. As a results, unless explicitly told otherwise, these languages assume that the current CPU is the only thing that is reading or writing memory. This in turn means that these languages' compilers' optimizers are ready, willing, and oh so able to make dramatic changes to the order, number, and sizes of memory references that your program executes. In fact, the reordering carried out by hardware can seem quite tame by comparison.

This section will help you tame your compiler, thus avoiding a great deal of compile-time consternation. Section 15.3.1 describes how to keep the compiler from destructively optimizing your code's memory references, Section 15.3.2 describes how to protect address and data dependencies, and finally, Section 15.3.3 describes how to protect those delicate control dependencies.

15.3.1 Memory-Reference Restrictions

As noted in Section 4.3.4, unless told otherwise, compilers assume that nothing else is affecting the variables that the code is accessing. Furthermore, this assumption is not simply some design error, but is instead enshrined in various standards.[11] It is worth summarizing this material in preparation for the following sections.

Plain accesses, as in plain-access C-language assignment statements such as "`r1 = a`" or "`b = 1`" are subject to the shared-variable shenanigans described in Section 4.3.4.1. Ways of avoiding these shenanigans are described in Sections 4.3.4.2–4.3.4.4 starting on page 43sec:toolsoftrade:Avoiding Data Races:

1. Plain accesses can tear, for example, the compiler could choose to access an eight-byte pointer one byte at a time. Tearing of aligned machine-sized accesses can be prevented by using `READ_ONCE()` and `WRITE_ONCE()`.

2. Plain loads can fuse, for example, if the results of an earlier load from that same object are still in a

[11] Or perhaps it is a standardized design error.

www.dbooks.org

machine register, the compiler might opt to reuse the value in that register instead of reloading from memory. Load fusing can be prevented by using READ_ONCE() or by enforcing ordering between the two loads using barrier(), smp_rmb(), and other means shown in Table 15.3.

3. Plain stores can fuse, so that a store can be omitted entirely if there is a later store to that same variable. Store fusing can be prevented by using WRITE_ONCE() or by enforcing ordering between the two stores using barrier(), smp_wmb(), and other means shown in Table 15.3.

4. Plain accesses can be reordered in surprising ways by modern optimizing compilers. This reordering can be prevented by enforcing ordering as called out above.

5. Plain loads can be invented, for example, register pressure might cause the compiler to discard a previously loaded value from its register, and then reload it later on. Invented loads can be prevented by using READ_ONCE() or by enforcing ordering as called out above between the load and a later use of its value using barrier().

6. Stores can be invented before a plain store, for example, by using the stored-to location as temporary storage. This can be prevented by use of WRITE_ONCE().

Quick Quiz 15.30: Why not place a barrier() call immediately before a plain store to prevent the compiler from inventing stores? ■

Please note that all of these shared-memory shenanigans can instead be avoided by avoiding data races on plain accesses, as described in Section 4.3.4.4. After all, if there are no data races, then each and every one of the compiler optimizations mentioned above is perfectly safe. But for code containing data races, this list is subject to change without notice as compiler optimizations continue becoming increasingly aggressive.

In short, use of READ_ONCE(), WRITE_ONCE(), barrier(), volatile, and other primitives called out in Table 15.3 on page 301 are valuable tools in preventing the compiler from optimizing your parallel algorithm out of existence. Compilers are starting to provide other mechanisms for avoiding load and store tearing, for example, memory_order_relaxed atomic loads and stores, however, work is still needed [Cor16b]. In addition, compiler issues aside, volatile is still needed to avoid fusing and invention of accesses, including C11 atomic accesses.

Please note that, it is possible to overdo use of READ_ONCE() and WRITE_ONCE(). For example, if you have prevented a given variable from changing (perhaps by holding the lock guarding all updates to that variable), there is no point in using READ_ONCE(). Similarly, if you have prevented any other CPUs or threads from reading a given variable (perhaps because you are initializing that variable before any other CPU or thread has access to it), there is no point in using WRITE_ONCE(). However, in my experience, developers need to use things like READ_ONCE() and WRITE_ONCE() more often than they think that they do, and the overhead of unnecessary uses is quite low. In contrast, the penalty for failing to use them when needed can be quite high.

15.3.2 Address- and Data-Dependency Difficulties

Compilers do not understand either address or data dependencies, although there are efforts underway to teach them, or at the very least, standardize the process of teaching them [MWB+17, MRP+17]. In the meantime, it is necessary to be very careful in order to prevent your compiler from breaking your dependencies.

15.3.2.1 Give your dependency chain a good start

The load that heads your dependency chain must use proper ordering, for example rcu_dereference() or READ_ONCE(). Failure to follow this rule can have serious side effects:

1. On DEC Alpha, a dependent load might not be ordered with the load heading the dependency chain, as described in Section 15.5.1.

2. If the load heading the dependency chain is a C11 non-volatile memory_order_relaxed load, the compiler could omit the load, for example, by using a value that it loaded in the past.

3. If the load heading the dependency chain is a plain load, the compiler can omit the load, again by using a value that it loaded in the past. Worse yet, it could load twice instead of once, so that different parts of your code use different values—and compilers really do this, especially when under register pressure.

Listing 15.26: Breakable Dependencies With Comparisons

```
1 int reserve_int;
2 int *gp;
3 int *p;
4
5 p = rcu_dereference(gp);
6 if (p == &reserve_int)
7   handle_reserve(p);
8 do_something_with(*p); /* buggy! */
```

Listing 15.27: Broken Dependencies With Comparisons

```
1 int reserve_int;
2 int *gp;
3 int *p;
4
5 p = rcu_dereference(gp);
6 if (p == &reserve_int) {
7   handle_reserve(&reserve_int);
8   do_something_with(reserve_int); /* buggy! */
9 } else {
10  do_something_with(*p); /* OK! */
11 }
```

4. The value loaded by the head of the dependency chain must be a pointer. In theory, yes, you could load an integer, perhaps to use it as an array index. In practice, the compiler knows too much about integers, and thus has way too many opportunities to break your dependency chain [MWB+17].

15.3.2.2 Avoid arithmetic dependency breakage

Although it is just fine to do some arithmetic operations on a pointer in your dependency chain, you need to be careful to avoid giving the compiler too much information. After all, if the compiler learns enough to determine the exact value of the pointer, it can use that exact value instead of the pointer itself. As soon as the compiler does that, the dependency is broken and all ordering is lost.

1. Although it is permissible to compute offsets from a pointer, these offsets must not result in total cancellation. For example, given a `char` pointer `cp`, `cp-(uintptr_t)cp)` will cancel and can allow the compiler to break your dependency chain. On the other hand, canceling offset values with each other is perfectly safe and legal. For example, if `a` and `b` are equal, `cp+a-b` is an identity function, including preserving the dependency.

2. Comparisons can break dependencies. Listing 15.26 shows how this can happen. Here global pointer `gp` points to a dynamically allocated integer, but if memory is low, it might instead point to the `reserve_int` variable. This `reserve_int` case might need special handling, as shown on lines 6 and 7 of the listing. But the compiler could reasonably transform this code into the form shown in Listing 15.27, especially on systems where instructions with absolute addresses run faster than instructions using addresses supplied in registers. However, there is clearly no ordering between the pointer load on line 5 and the dereference on line 8. Please note that this is simply an example: There are a great many other ways to break dependency chains with comparisons.

Quick Quiz 15.31: Why can't you simply dereference the pointer before comparing it to `&reserve_int` on line 6 of Listing 15.26? ■

Quick Quiz 15.32: But it should be safe to compare two pointer variables, right? After all, the compiler doesn't know the value of either, so how can it possibly learn anything from the comparison? ■

Note that a series of inequality comparisons might, when taken together, give the compiler enough information to determine the exact value of the pointer, at which point the dependency is broken. Furthermore, the compiler might be able to combine information from even a single inequality comparison with other information to learn the exact value, again breaking the dependency. Pointers to elements in arrays are especially susceptible to this latter form of dependency breakage.

15.3.2.3 Safe comparison of dependent pointers

It turns out that there are several safe ways to compare dependent pointers:

1. Comparisons against the `NULL` pointer. In this case, all the compiler can learn is that the pointer is `NULL`, in which case you are not allowed to dereference it anyway.

2. The dependent pointer is never dereferenced, whether before or after the comparison.

3. The dependent pointer is compared to a pointer that references objects that were last modified a very long time ago, where the only unconditionally safe value of "a very long time ago" is "at compile time". The key point is that something other than the address or data dependency guarantees ordering.

4. Comparisons between two pointers, each of which carries an appropriate dependency. For example, you have a pair of pointers, each carrying a dependency,

www.dbooks.org

Listing 15.28: Broken Dependencies With Pointer Comparisons

```
1 struct foo {
2   int a;
3   int b;
4   int c;
5 };
6 struct foo *gp1;
7 struct foo *gp2;
8
9 void updater(void)
10 {
11   struct foo *p;
12
13   p = malloc(sizeo(*p));
14   BUG_ON(!p);
15   p->a = 42;
16   p->b = 43;
17   p->c = 44;
18   rcu_assign_pointer(gp1, p);
19   WRITE_ONCE(p->b, 143);
20   WRITE_ONCE(p->c, 144);
21   rcu_assign_pointer(gp2, p);
22 }
23
24 void reader(void)
25 {
26   struct foo *p;
27   struct foo *q;
28   int r1, r2 = 0;
29
30   p = rcu_dereference(gp2);
31   if (p == NULL)
32     return;
33   r1 = READ_ONCE(p->b);
34   q = rcu_dereference(gp1);
35   if (p == q) {
36     r2 = READ_ONCE(p->c);
37   }
38   do_something_with(r1, r2);
39 }
```

to data structures each containing a lock, and you want to avoid deadlock by acquiring the locks in address order.

5. The comparison is not-equal, and the compiler does not have enough other information to deduce the value of the pointer carrying the dependency.

Pointer comparisons can be quite tricky, and so it is well worth working through the example shown in Listing 15.28. This example uses a simple `struct foo` shown on lines 1–5 and two global pointers, `gp1` and `gp2`, shown on lines 6 and 7, respectively. This example uses two threads, namely `updater()` on lines 9–22 and `reader()` on lines 24–39.

The `updater()` thread allocates memory on line 13, and complains bitterly on line 14 if none is available. Lines 15–17 initialize the newly allocated structure, and then line 18 assigns the pointer to `gp1`. Lines 19 and 20 then update two of the structure's fields, and does so `after` line 18 has made those fields visible to readers. Please note that unsynchronized update of reader-visible fields often constitutes a bug. Although there are legitimate use cases doing just this, such use cases require more care than is exercised in this example.

Finally, line 21 assigns the pointer to `gp2`.

The `reader()` thread first fetches `gp2` on line 30, with lines 31 and 32 checking for `NULL` and returning if so. Line 33 fetches field `->b` and line 34 fetches `gp1`. If line 35 sees that the pointers fetched on lines 30 and 34 are equal, line 36 fetches `p->c`. Note that line 36 uses pointer `p` fetched on line 30, not pointer `q` fetched on line 34.

But this difference might not matter. An equals comparison on line 35 might lead the compiler to (incorrectly) conclude that both pointers are equivalent, when in fact they carry different dependencies. This means that the compiler might well transform line 36 to instead be `r2 = q->c`, which might well cause the value 44 to be loaded instead of the expected value 144.

Quick Quiz 15.33: But doesn't the condition in line 35 supply a control dependency that would keep line 36 ordered after line 34? ■

In short, great care is required to ensure that dependency chains in your source code are still dependency chains in the compiler-generated assembly code.

15.3.3 Control-Dependency Calamities

Control dependencies are especially tricky because current compilers do not understand them and can easily break them. The rules and examples in this section are intended to help you prevent your compiler's ignorance from breaking your code.

A load-load control dependency requires a full read memory barrier, not simply a data dependency barrier. Consider the following bit of code:

```
1 q = READ_ONCE(x);
2 if (q) {
3   <data dependency barrier>
4   q = READ_ONCE(y);
5 }
```

This will not have the desired effect because there is no actual data dependency, but rather a control dependency that the CPU may short-circuit by attempting to predict the outcome in advance, so that other CPUs see the load from `y` as having happened before the load from `x`. In such a case what's actually required is:

```
1 q = READ_ONCE(x);
2 if (q) {
```

```
  <read barrier>
  q = READ_ONCE(y);
}
```

However, stores are not speculated. This means that ordering *is* provided for load-store control dependencies, as in the following example:

```
q = READ_ONCE(x);
if (q)
  WRITE_ONCE(y, 1);
```

Control dependencies pair normally with other types of ordering operations. That said, please note that neither `READ_ONCE()` nor `WRITE_ONCE()` are optional! Without the `READ_ONCE()`, the compiler might fuse the load from `x` with other loads from `x`. Without the `WRITE_ONCE()`, the compiler might fuse the store to `y` with other stores to `y`. Either can result in highly counter-intuitive effects on ordering.

Worse yet, if the compiler is able to prove (say) that the value of variable `x` is always non-zero, it would be well within its rights to optimize the original example by eliminating the "`if`" statement as follows:

```
q = READ_ONCE(x);
WRITE_ONCE(y, 1); /* BUG: CPU can reorder!!! */
```

It is tempting to try to enforce ordering on identical stores on both branches of the "`if`" statement as follows:

```
q = READ_ONCE(x);
if (q) {
  barrier();
  WRITE_ONCE(y, 1);
  do_something();
} else {
  barrier();
  WRITE_ONCE(y, 1);
  do_something_else();
}
```

Unfortunately, current compilers will transform this as follows at high optimization levels:

```
q = READ_ONCE(x);
barrier();
WRITE_ONCE(y, 1);  /* BUG: No ordering!!! */
if (q) {
  do_something();
} else {
  do_something_else();
}
```

Now there is no conditional between the load from `x` and the store to `y`, which means that the CPU is within its rights to reorder them: The conditional is absolutely required, and must be present in the assembly code even after all compiler optimizations have been applied. Therefore, if you need ordering in this example, you need explicit memory-ordering operations, for example, a release store:

```
q = READ_ONCE(x);
if (q) {
  smp_store_release(&y, 1);
  do_something();
} else {
  smp_store_release(&y, 1);
  do_something_else();
}
```

The initial `READ_ONCE()` is still required to prevent the compiler from guessing the value of `x`. In addition, you need to be careful what you do with the local variable `q`, otherwise the compiler might be able to guess its value and again remove the needed conditional. For example:

```
q = READ_ONCE(x);
if (q % MAX) {
  WRITE_ONCE(y, 1);
  do_something();
} else {
  WRITE_ONCE(y, 2);
  do_something_else();
}
```

If `MAX` is defined to be 1, then the compiler knows that `(q%MAX)` is equal to zero, in which case the compiler is within its rights to transform the above code into the following:

```
q = READ_ONCE(x);
WRITE_ONCE(y, 2);
do_something_else();
```

Given this transformation, the CPU is not required to respect the ordering between the load from variable `x` and the store to variable `y`. It is tempting to add a `barrier()` to constrain the compiler, but this does not help. The conditional is gone, and the `barrier()` won't bring it back. Therefore, if you are relying on this ordering, you should make sure that `MAX` is greater than one, perhaps as follows:

```
q = READ_ONCE(x);
BUILD_BUG_ON(MAX <= 1);
if (q % MAX) {
  WRITE_ONCE(y, 1);
  do_something();
} else {
  WRITE_ONCE(y, 2);
  do_something_else();
}
```


www.dbooks.org

Please note once again that the stores to y differ. If they were identical, as noted earlier, the compiler could pull this store outside of the "if" statement.

You must also avoid excessive reliance on boolean short-circuit evaluation. Consider this example:

```
q = READ_ONCE(x);
if (q || 1 > 0)
  WRITE_ONCE(y, 1);
```

Because the first condition cannot fault and the second condition is always true, the compiler can transform this example as following, defeating control dependency:

```
q = READ_ONCE(x);
WRITE_ONCE(y, 1);
```

This example underscores the need to ensure that the compiler cannot out-guess your code. More generally, although READ_ONCE() does force the compiler to actually emit code for a given load, it does not force the compiler to use the value loaded.

In addition, control dependencies apply only to the then-clause and else-clause of the if-statement in question. In particular, it does not necessarily apply to code following the if-statement:

```
q = READ_ONCE(x);
if (q) {
  WRITE_ONCE(y, 1);
} else {
  WRITE_ONCE(y, 2);
}
WRITE_ONCE(z, 1);  /* BUG: No ordering. */
```

It is tempting to argue that there in fact is ordering because the compiler cannot reorder volatile accesses and also cannot reorder the writes to y with the condition. Unfortunately for this line of reasoning, the compiler might compile the two writes to y as conditional-move instructions, as in this fanciful pseudo-assembly language:

```
ld r1,x
cmp r1,$0
cmov,ne r4,$1
cmov,eq r4,$2
st r4,y
st $1,z
```

A weakly ordered CPU would have no dependency of any sort between the load from x and the store to z. The control dependencies would extend only to the pair of cmov instructions and the store depending on them. In short, control dependencies apply only to the stores in the "then" and "else" of the "if" in question (including functions invoked by those two clauses), and not necessarily to code following that "if".

Listing 15.29: LB Litmus Test With Control Dependency

```
C C-LB+o-cgt-o+o-cgt-o

{}

P0(int *x, int *y)
{
  int r1;

  r1 = READ_ONCE(*x);
  if (r1 > 0)
    WRITE_ONCE(*y, 1);
}

P1(int *x, int *y)
{
  int r2;

  r2 = READ_ONCE(*y);
  if (r2 > 0)
    WRITE_ONCE(*x, 1);
}

exists (0:r1=1 /\ 1:r2=1)
```

Finally, control dependencies do *not* provide cumulativity.[12] This is demonstrated by two related litmus tests, namely Listings 15.29 and 15.30 with the initial values of x and y both being zero.

The exists clause in the two-thread example of Listing 15.29 (C-LB+o-cgt-o+o-cgt-o.litmus) will never trigger. If control dependencies guaranteed cumulativity (which they do not), then adding a thread to the example as in Listing 15.30 (C-WWC+o-cgt-o+o-cgt-o+o.litmus) would guarantee the related exists clause never to trigger.

But because control dependencies do *not* provide cumulativity, the exists clause in the three-thread litmus test can trigger. If you need the three-thread example to provide ordering, you will need smp_mb() between the load and store in P0(), that is, just before or just after the "if" statements. Furthermore, the original two-thread example is very fragile and should be avoided.

Quick Quiz 15.34: Can't you instead add an smp_mb() to P1() in Listing 15.30? ■

The following list of rules summarizes the lessons of this section:

1. Compilers do not understand control dependencies, so it is your job to make sure that the compiler cannot break your code.

[12] Refer to Section 15.2.7.1 for the meaning of cumulativity.

Listing 15.30: WWC Litmus Test With Control Dependency (Cumulativity?)

```
C C-WWC+o-cgt-o+o-cgt-o+o

{}

P0(int *x, int *y)
{
  int r1;

  r1 = READ_ONCE(*x);
  if (r1 > 0)
    WRITE_ONCE(*y, 1);
}

P1(int *x, int *y)
{
  int r2;

  r2 = READ_ONCE(*y);
  if (r2 > 0)
    WRITE_ONCE(*x, 1);
}

P2(int *x)
{
  WRITE_ONCE(*x, 2);
}

exists (0:r1=2 /\ 1:r2=1 /\ x=2)
```

2. Control dependencies can order prior loads against later stores. However, they do *not* guarantee any other sort of ordering: Not prior loads against later loads, nor prior stores against later anything. If you need these other forms of ordering, use `smp_rmb()`, `smp_wmb()`, or, in the case of prior stores and later loads, `smp_mb()`.

3. If both legs of the "`if`" statement begin with identical stores to the same variable, then the control dependency will not order those stores, If ordering is needed, precede both of them with `smp_mb()` or use `smp_store_release()`. Please note that it is *not* sufficient to use `barrier()` at beginning of each leg of the "`if`" statement because, as shown by the example above, optimizing compilers can destroy the control dependency while respecting the letter of the `barrier()` law.

4. Control dependencies require at least one run-time conditional between the prior load and the subsequent store, and this conditional must involve the prior load. If the compiler is able to optimize the conditional away, it will have also optimized away the ordering. Careful use of `READ_ONCE()` and `WRITE_ONCE()` can help to preserve the needed conditional.

5. Control dependencies require that the compiler avoid reordering the dependency into nonexistence. Careful use of `READ_ONCE()`, `atomic_read()`, or `atomic64_read()` can help to preserve your control dependency.

6. Control dependencies apply only to the "`then`" and "`else`" of the "`if`" containing the control dependency, including any functions that these two clauses call. Control dependencies do *not* apply to code following the end of the "`if`" statement containing the control dependency.

7. Control dependencies pair normally with other types of memory-ordering operations.

8. Control dependencies do *not* provide cumulativity. If you need cumulativity, use something that provides it, such as `smp_store_release()` or `smp_mb()`.

Again, many popular languages were designed with single-threaded use in mind. Successful multithreaded use of these languages requires you to pay special attention to your memory references and dependencies.

15.4 Higher-Level Primitives

> Method will teach you to win time.
>
> *Johann Wolfgang von Goethe*

The answer to one of the quick quizzes in Section 12.3.1 demonstrated exponential speedups due to verifying programs modeled at higher levels of abstraction. This section will look into how higher levels of abstraction can also provide a deeper understanding of the synchronization primitives themselves. Section 15.4.1 takes a look at memory allocation and Section 15.4.2 digs more deeply into RCU.

15.4.1 Memory Allocation

Section 6.4.3.2 touched upon memory allocation, and this section expands upon the relevant memory-ordering issues.

The key requirement is that any access executed on a given block of memory before freeing that block must be ordered before any access executed after that same block is reallocated. It would after all be a cruel and unusual memory-allocator bug if a store preceding the free were to

www.dbooks.org

be reordered after another store following the reallocation! However, it would also be cruel and unusual to require developers to use `READ_ONCE()` and `WRITE_ONCE()` to access dynamically allocated memory. Full ordering must therefore be provided for plain accesses, in spite of all the shared-variable shenanigans called out in Section 4.3.4.1.

Of course, each CPU sees its own accesses in order and the compiler always has fully accounted for intra-CPU shenanigans. These facts are what enables the lockless fastpaths in `memblock_alloc()` and `memblock_free()`, which are shown in Listings 6.10 and 6.11, respectively. However, this also why the developer is responsible for providing appropriate ordering (for example, by using `smp_store_release()`) when publishing a pointer to a newly allocated block of memory. After all, in the CPU-local case, the allocator has not necessarily provided any ordering.

However, the allocator must provide ordering when rebalancing its per-thread pools. This ordering is provided by the calls to `spin_lock()` and `spin_unlock()` from `memblock_alloc()` and `memblock_free()`. For any block that has migrated from one thread to another, the old thread will have executed `spin_unlock(&globalmem.mutex)` after placing the block in the `globalmem` pool, and the new thread will have executed `spin_lock(&globalmem.mutex)` before moving that block to its per-thread pool. This `spin_unlock()` and `spin_lock()` ensures that both the old and new threads see the old thread's accesses as having happened before those of the new thread.

Quick Quiz 15.35: But doesn't PowerPC have weak unlock-lock ordering properties within the Linux kernel, allowing a write before the unlock to be reordered with a read after the lock? ■

Therefore, the ordering required by conventional uses of memory allocation can be provided solely by non-fastpath locking, allowing the fastpath to remain synchronization-free.

15.4.2 RCU

As described in Section 9.5.2, the fundamental property of RCU grace periods is this straightforward two-part guarantee: (1) If any part of a given RCU read-side critical section precedes the beginning of a given grace period, then the entirety of that critical section precedes the end of that grace period. (2) If any part of a given RCU read-side critical section follows the end of a given grace period, then the entirety of that critical section follows

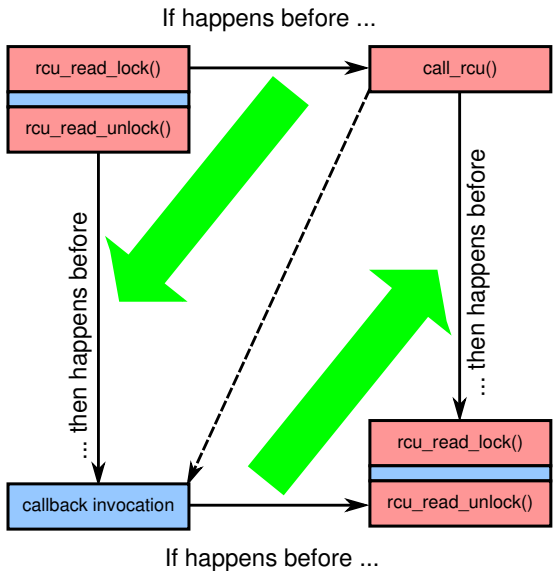


Figure 15.13: RCU Grace-Period Ordering Guarantees

Listing 15.31: RCU Fundamental Property

```
1  C C-SB+o-rcusync-o+rl-o-o-rul
2
3  {}
4
5  P0(uintptr_t *x0, uintptr_t *x1)
6  {
7    WRITE_ONCE(*x0, 2);
8    synchronize_rcu();
9    uintptr_t r2 = READ_ONCE(*x1);
10 }
11
12 P1(uintptr_t *x0, uintptr_t *x1)
13 {
14   rcu_read_lock();
15   WRITE_ONCE(*x1, 2);
16   uintptr_t r2 = READ_ONCE(*x0);
17   rcu_read_unlock();
18 }
19
20 exists (1:r2=0 /\ 0:r2=0)
```

the beginning of that grace period. These guarantees are summarized in Figure 15.13, where the grace period is denoted by the dashed arrow between the `call_rcu()` invocation in the upper right and the corresponding RCU callback invocation in the lower left.[13]

In short, an RCU read-side critical section is guaranteed never to completely overlap an RCU grace period, as demonstrated by Listing 15.31 (`C-SB+o-rcusync-o+rl-o-o-rul.litmus`). Either or neither of the `r2` registers can have the final value of zero, but at least one

[13] For more detail, please see Figures 9.11–9.13 starting on page 142.

Listing 15.32: RCU Fundamental Property and Reordering

```
C C-SB+o-rcusync-o+i-rl-o-o-rul

{}

P0(uintptr_t *x0, uintptr_t *x1)
{
  WRITE_ONCE(*x0, 2);
  synchronize_rcu();
  uintptr_t r2 = READ_ONCE(*x1);
}

P1(uintptr_t *x0, uintptr_t *x1)
{
  rcu_read_lock();
  uintptr_t r2 = READ_ONCE(*x0);
  WRITE_ONCE(*x1, 2);
  rcu_read_unlock();
}

exists (1:r2=0 /\ 0:r2=0)
```

Listing 15.33: RCU Readers Provide No Lock-Like Ordering

```
C C-LB+rl-o-o-rul+rl-o-o-rul

{}

P0(uintptr_t *x0, uintptr_t *x1)
{
  rcu_read_lock();
  uintptr_t r1 = READ_ONCE(*x0);
  WRITE_ONCE(*x1, 1);
  rcu_read_unlock();
}

P1(uintptr_t *x0, uintptr_t *x1)
{
  rcu_read_lock();
  uintptr_t r1 = READ_ONCE(*x1);
  WRITE_ONCE(*x0, 1);
  rcu_read_unlock();
}

exists (0:r1=1 /\ 1:r1=1)
```

of them must be non-zero (that is, the cycle identified by the `exists` clause is prohibited), courtesy of RCU's fundamental grace-period guarantee, as can be seen by running `herd` on this litmus test. Note that this guarantee is insensitive to the ordering of the accesses within `P1()`'s critical section, so the litmus test shown in Listing 15.32[14] also forbids this same cycle.

However, this definition is incomplete, as can be seen from the following list of questions:[15]

1. What ordering is provided by `rcu_read_lock()` and `rcu_read_unlock()`, independent of RCU grace periods?
2. What ordering is provided by `synchronize_rcu()` and `synchronize_rcu_expedited()`, independent of RCU read-side critical sections?
3. If the entirety of a given RCU read-side critical section precedes the end of a given RCU grace period, what about accesses preceding that critical section?
4. If the entirety of a given RCU read-side critical section follows the beginning of a given RCU grace period, what about accesses following that critical section?
5. What happens in situations involving more than one RCU read-side critical section and/or more than one RCU grace period?
6. What happens when RCU is mixed with other memory-ordering mechanisms?

These questions are addressed in the following sections.

15.4.2.1 RCU Read-Side Ordering

On their own, RCU's read-side primitives `rcu_read_lock()` and `rcu_read_unlock()` provide no ordering whatsoever. In particular, despite their names, they do not act like locks, as can be seen in Listing 15.33 (`C-LB+rl-o-o-rul+rl-o-o-rul.litmus`). This litmus test's cycle is allowed: Both instances of the `r1` register can have final values of 1.

Nor do these primitives have barrier-like ordering properties, at least not unless there is a grace period in the mix, as can be seen in Listing 15.34 (`C-LB+o-rl-rul-o+o-rl-rul-o.litmus`). This litmus test's cycle is also allowed. (Try it!)

Of course, lack of ordering in both these litmus tests should be absolutely no surprise, given that both `rcu_read_lock()` and `rcu_read_unlock()` are no-ops in the QSBR implementation of RCU.

15.4.2.2 RCU Update-Side Ordering

In contrast with RCU readers, the RCU update-side functions `synchronize_rcu()` and `synchronize_rcu_expedited()` provide memory ordering at least as strong as `smp_mb()`, as can be seen by running `herd` on the litmus test shown in Listing 15.35. This test's cycle is prohibited, just as it would with `smp_mb()`. This should be no surprise given the information presented in Table 15.3.

[14] Dependencies can of course limit the ability to reorder accesses within RCU read-side critical sections.

[15] Several of which were introduced to Paul by Jade Alglave during early work on LKMM, and a few more of which came from other LKMM participants [AMM+18].

www.dbooks.org

Listing 15.34: RCU Readers Provide No Barrier-Like Ordering

```
C C-LB+o-rl-rul-o+o-rl-rul-o

{}

P0(uintptr_t *x0, uintptr_t *x1)
{
  uintptr_t r1 = READ_ONCE(*x0);
  rcu_read_lock();
  rcu_read_unlock();
  WRITE_ONCE(*x1, 1);
}

P1(uintptr_t *x0, uintptr_t *x1)
{
  uintptr_t r1 = READ_ONCE(*x1);
  rcu_read_lock();
  rcu_read_unlock();
  WRITE_ONCE(*x0, 1);
}

exists (0:r1=1 /\ 1:r1=1)
```

Listing 15.35: RCU Updaters Provide Full Ordering

```
C C-SB+o-rcusync-o+o-rcusync-o

{}

P0(uintptr_t *x0, uintptr_t *x1)
{
  WRITE_ONCE(*x0, 2);
  synchronize_rcu();
  uintptr_t r2 = READ_ONCE(*x1);
}

P1(uintptr_t *x0, uintptr_t *x1)
{
  WRITE_ONCE(*x1, 2);
  synchronize_rcu();
  uintptr_t r2 = READ_ONCE(*x0);
}

exists (1:r2=0 /\ 0:r2=0)
```

Listing 15.36: What Happens Before RCU Readers?

```
C C-SB+o-rcusync-o+o-rl-o-rul

{}

P0(uintptr_t *x0, uintptr_t *x1)
{
  WRITE_ONCE(*x0, 2);
  synchronize_rcu();
  uintptr_t r2 = READ_ONCE(*x1);
}

P1(uintptr_t *x0, uintptr_t *x1)
{
  WRITE_ONCE(*x1, 2);
  rcu_read_lock();
  uintptr_t r2 = READ_ONCE(*x0);
  rcu_read_unlock();
}

exists (1:r2=0 /\ 0:r2=0)
```

15.4.2.3 RCU Readers: Before and After

Before reading this section, it would be well to reflect on the distinction between guarantees that are available and guarantees that maintainable software should rely on. Keeping that firmly in mind, this section presents a few of the more exotic RCU guarantees.

Listing 15.36 (`C-SB+o-rcusync-o+o-rl-o-rul.litmus`) shows a litmus test similar to that in Listing 15.31, but with the RCU reader's first access preceding the RCU read-side critical section, rather than the more conventional (and maintainable!) approach of being contained within it. Perhaps surprisingly, running `herd` on this litmus test gives the same result as for that in Listing 15.31: The cycle is forbidden.

Why would this be the case?

Because both of `P1()`'s accesses are volatile, as discussed in Section 4.3.4.2, the compiler is not permitted to reorder them. This means that the code emitted for `P1()`'s `WRITE_ONCE()` will precede that of `P1()`'s `READ_ONCE()`. Therefore, RCU implementations that place memory-barrier instructions in `rcu_read_lock()` and `rcu_read_unlock()` will preserve the ordering of `P1()`'s two accesses all the way down to the hardware level. On the other hand, RCU implementations that rely on interrupt-based state machines will also fully preserve this ordering *relative to the grace period* due to the fact that interrupts take place at a precise location in the execution of the interrupted code.

This in turn means that if the `WRITE_ONCE()` follows the end of a given RCU grace period, then the accesses within the RCU read-side critical section must follow the beginning of that same grace period. Similarly, if the `READ_ONCE()` precedes the beginning of the grace period,

Listing 15.37: What Happens After RCU Readers?

```
C C-SB+o-rcusync-o+rl-o-rul-o

{}

P0(uintptr_t *x0, uintptr_t *x1)
{
  WRITE_ONCE(*x0, 2);
  synchronize_rcu();
  uintptr_t r2 = READ_ONCE(*x1);
}

P1(uintptr_t *x0, uintptr_t *x1)
{
  rcu_read_lock();
  WRITE_ONCE(*x1, 2);
  rcu_read_unlock();
  uintptr_t r2 = READ_ONCE(*x0);
}

exists (1:r2=0 /\ 0:r2=0)
```

Listing 15.38: What Happens With Empty RCU Readers?

```
C C-SB+o-rcusync-o+o-rl-rul-o

{}

P0(uintptr_t *x0, uintptr_t *x1)
{
  WRITE_ONCE(*x0, 2);
  synchronize_rcu();
  uintptr_t r2 = READ_ONCE(*x1);
}

P1(uintptr_t *x0, uintptr_t *x1)
{
  WRITE_ONCE(*x1, 2);
  rcu_read_lock();
  rcu_read_unlock();
  uintptr_t r2 = READ_ONCE(*x0);
}

exists (1:r2=0 /\ 0:r2=0)
```

everything within *and preceding* that critical section must precede the end of that same grace period.

Listing 15.37 (`C-SB+o-rcusync-o+rl-o-rul-o.litmus`) is similar, but instead looks at accesses after the RCU read-side critical section. This test's cycle is also forbidden, as can be checked with the `herd` tool. The reasoning is similar to that for Listing 15.36, and is left as an exercise for the reader.

Listing 15.38 (`C-SB+o-rcusync-o+o-rl-rul-o.litmus`) takes things one step farther, moving `P1()`'s `WRITE_ONCE()` to precede the RCU read-side critical section and moving `P1()`'s `READ_ONCE()` to follow it, resulting in an empty RCU read-side critical section.

Perhaps surprisingly, despite the empty critical section, RCU nevertheless still manages to forbid the cycle. This can again be checked using the `herd` tool. Furthermore, the reasoning is once again similar to that for Listing 15.36,

Listing 15.39: What Happens With No RCU Readers?

```
C C-SB+o-rcusync-o+o-o

{}

P0(uintptr_t *x0, uintptr_t *x1)
{
  WRITE_ONCE(*x0, 2);
  synchronize_rcu();
  uintptr_t r2 = READ_ONCE(*x1);
}

P1(uintptr_t *x0, uintptr_t *x1)
{
  WRITE_ONCE(*x1, 2);
  uintptr_t r2 = READ_ONCE(*x0);
}

exists (1:r2=0 /\ 0:r2=0)
```

Recapping, if `P1()`'s `WRITE_ONCE()` follows the end of a given grace period, then `P1()`'s RCU read-side critical section—and everything following it—must follow the beginning of that same grace period. Similarly, if `P1()`'s `READ_ONCE()` precedes the beginning of a given grace period, then `P1()`'s RCU read-side critical section—and everything preceding it—must precede the end of that same grace period. In both cases, the critical section's emptiness is irrelevant.

Quick Quiz 15.36: Wait a minute! In QSBR implementations of RCU, no code is emitted for `rcu_read_lock()` and `rcu_read_unlock()`. This means that the RCU read-side critical section in Listing 15.38 isn't just empty, it is completely nonexistent!!! So how can something that doesn't exist at all possibly have any effect whatsoever on ordering??? ■

This situation leads to the question of what happens if `rcu_read_lock()` and `rcu_read_unlock()` are omitted entirely, as shown in Listing 15.39 (`C-SB+o-rcusync-o+o-o.litmus`). As can be checked with `herd`, this litmus test's cycle is allowed, that is, both instances of `r2` can have final values of zero.

This might seem strange in light of the fact that empty RCU read-side critical sections can provide ordering. And it is true that a QSBR implementations of RCU would in fact forbid this outcome. However, RCU also has non-QSBR implementations, and these implementations have no way to enforce ordering in this case. Therefore, this litmus test's cycle is allowed.

Quick Quiz 15.37: Can `P1()`'s accesses be reordered in the litmus tests shown in Listings 15.36, 15.37, and 15.38 in the same way that they were reordered going from Listing 15.31 to Listing 15.32? ■

www.dbooks.org

Listing 15.40: One RCU Grace Period and Two Readers

```
1  C C-SB+o-rcusync-o+rl-o-o-rul+rl-o-o-rul
2
3  {}
4
5  P0(uintptr_t *x0, uintptr_t *x1)
6  {
7    WRITE_ONCE(*x0, 2);
8    synchronize_rcu();
9    uintptr_t r2 = READ_ONCE(*x1);
10 }
11
12 P1(uintptr_t *x1, uintptr_t *x2)
13 {
14   rcu_read_lock();
15   WRITE_ONCE(*x1, 2);
16   uintptr_t r2 = READ_ONCE(*x2);
17   rcu_read_unlock();
18 }
19
20 P2(uintptr_t *x2, uintptr_t *x0)
21 {
22   rcu_read_lock();
23   WRITE_ONCE(*x2, 2);
24   uintptr_t r2 = READ_ONCE(*x0);
25   rcu_read_unlock();
26 }
27
28 exists (2:r2=0 /\ 0:r2=0 /\ 1:r2=0)
```

15.4.2.4 Multiple RCU Readers and Updaters

Because `synchronize_rcu()` has stronger ordering semantics than does `smp_mb()`, no matter how many processes there are in an SB litmus test (such as Listing 15.35), placing `synchronize_rcu()` between each process's accesses prohibits the cycle. In addition, the cycle is prohibited in an SB test where one process uses `synchronize_rcu()` and the other uses `rcu_read_lock()` and `rcu_read_unlock()`, as exemplified by Listing 15.31. However, if both processes use `rcu_read_lock()` and `rcu_read_unlock()`, the cycle will be allowed, as exemplified by Listing 15.33.

Is it possible to say anything general about which RCU-protected litmus tests will be prohibited and which will be allowed? This section takes up this question.

More specifically, what if the litmus test has one RCU grace period and two RCU readers, as shown in Listing 15.40? The `herd` tool says that this cycle is allowed, but it would be good to know *why*.[16]

The key point is that the CPU is free to reorder `P1()`'s and `P2()`'s `WRITE_ONCE()` and `READ_ONCE()`. With that reordering, Figure 15.14 shows how the cycle forms:

[16] Especially given that Paul changed his mind several times about this particular litmus test when working with Jade Alglave to generalize RCU ordering semantics.

Listing 15.41: Two RCU Grace Periods and Two Readers

```
1  C C-SB+o-rcusync-o+o-rcusync-o+rl-o-o-rul+rl-o-o-rul
2
3  {}
4
5  P0(uintptr_t *x0, uintptr_t *x1)
6  {
7    WRITE_ONCE(*x0, 2);
8    synchronize_rcu();
9    uintptr_t r2 = READ_ONCE(*x1);
10 }
11
12 P1(uintptr_t *x1, uintptr_t *x2)
13 {
14   WRITE_ONCE(*x1, 2);
15   synchronize_rcu();
16   uintptr_t r2 = READ_ONCE(*x2);
17 }
18
19 P2(uintptr_t *x2, uintptr_t *x3)
20 {
21   rcu_read_lock();
22   WRITE_ONCE(*x2, 2);
23   uintptr_t r2 = READ_ONCE(*x3);
24   rcu_read_unlock();
25 }
26
27 P3(uintptr_t *x0, uintptr_t *x3)
28 {
29   rcu_read_lock();
30   WRITE_ONCE(*x3, 2);
31   uintptr_t r2 = READ_ONCE(*x0);
32   rcu_read_unlock();
33 }
34
35 exists (3:r2=0 /\ 0:r2=0 /\ 1:r2=0 /\ 2:r2=0)
```

1. `P0()`'s read from `x1` precedes `P1()`'s write, as depicted by the dashed arrow near the bottom of the diagram.
2. Because `P1()`'s write follows the end of `P0()`'s grace period, `P1()`'s read from `x2` cannot precede the beginning of `P0()`'s grace period.
3. `P1()`'s read from `x2` precedes `P2()`'s write.
4. Because `P2()`'s write to `x2` precedes the end of `P0()`'s grace period, it is completely legal for `P2()`'s read from `x0` to precede the beginning of `P0()`'s grace period.
5. Therefore, `P2()`'s read from `x0` can precede `P0()`'s write, thus allowing the cycle to form.

But what happens when another grace period is added? This situation is shown in Listing 15.41, an SB litmus test in which `P0()` and `P1()` have RCU grace periods and `P2()` and `P3()` have RCU readers. Again, the CPUs can reorder the accesses within RCU read-side critical sections, as shown in Figure 15.15. For this cycle to form, `P2()`'s critical section must end after `P1()`'s grace period

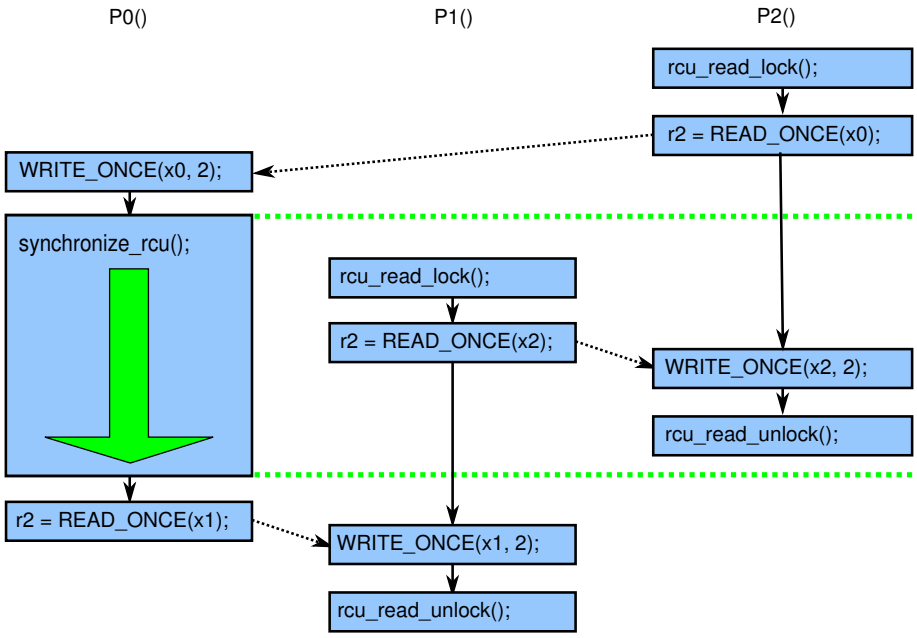


Figure 15.14: Cycle for One RCU Grace Period and Two RCU Readers

and P3()'s must end after the beginning of that same grace period, which happens to also be after the end of P0()'s grace period. Therefore, P3()'s critical section must start after the beginning of P0()'s grace period, which in turn means that P3()'s read from x0 cannot possibly precede P0()'s write. Therefore, the cycle is forbidden because RCU read-side critical sections cannot span full RCU grace periods.

However, a closer look at Figure 15.15 makes it clear that adding a third reader would allow the cycle. This is because this third reader could end before the end of P0()'s grace period, and thus start before the beginning of that same grace period. This in turn suggests the general rule, which is: In these sorts of RCU-only litmus tests, if there are at least as many RCU grace periods as there are RCU read-side critical sections, the cycle is forbidden.[17]

15.4.2.5 RCU and Other Ordering Mechanisms

But what about litmus tests that combine RCU with other ordering mechanisms?

The general rule is that it takes only one mechanism to forbid a cycle.

For example, refer back to Listing 15.33. Applying the general rule from the previous section, because this litmus test has two RCU read-side critical sections and no RCU grace periods, the cycle is allowed. But what if P0()'s WRITE_ONCE() is replaced by an smp_store_release() and P1()'s READ_ONCE() is replaced by an smp_load_acquire()?

RCU would still allow the cycle, but the release-acquire pair would forbid it. Because it only takes one mechanism to forbid a cycle, the release-acquire pair would prevail so that the cycle would be forbidden.

For another example, refer back to Listing 15.40. Because this litmus test has two RCU readers but only one grace period, its cycle is allowed. But suppose that an smp_mb() was placed between P1()'s pair of accesses. In this new litmus test, because of the addition of the smp_mb(), P2()'s as well as P1()'s critical sections would extend beyond the end of P0()'s grace period, which in turn would prevent P2()'s read from x0 from preceding P0()'s write, as depicted by the red dashed arrow in Figure 15.16. In this case, RCU and the full memory barrier work together to forbid the cycle, with RCU preserving ordering between P0() and both P1() and P2(), and

[17] Interestingly enough, Alan Stern proved that within the context of LKMM, the two-part fundamental property of RCU expressed in Section 9.5.2 actually implies this seemingly more general result, which is called the RCU axiom [AMM+18].

www.dbooks.org

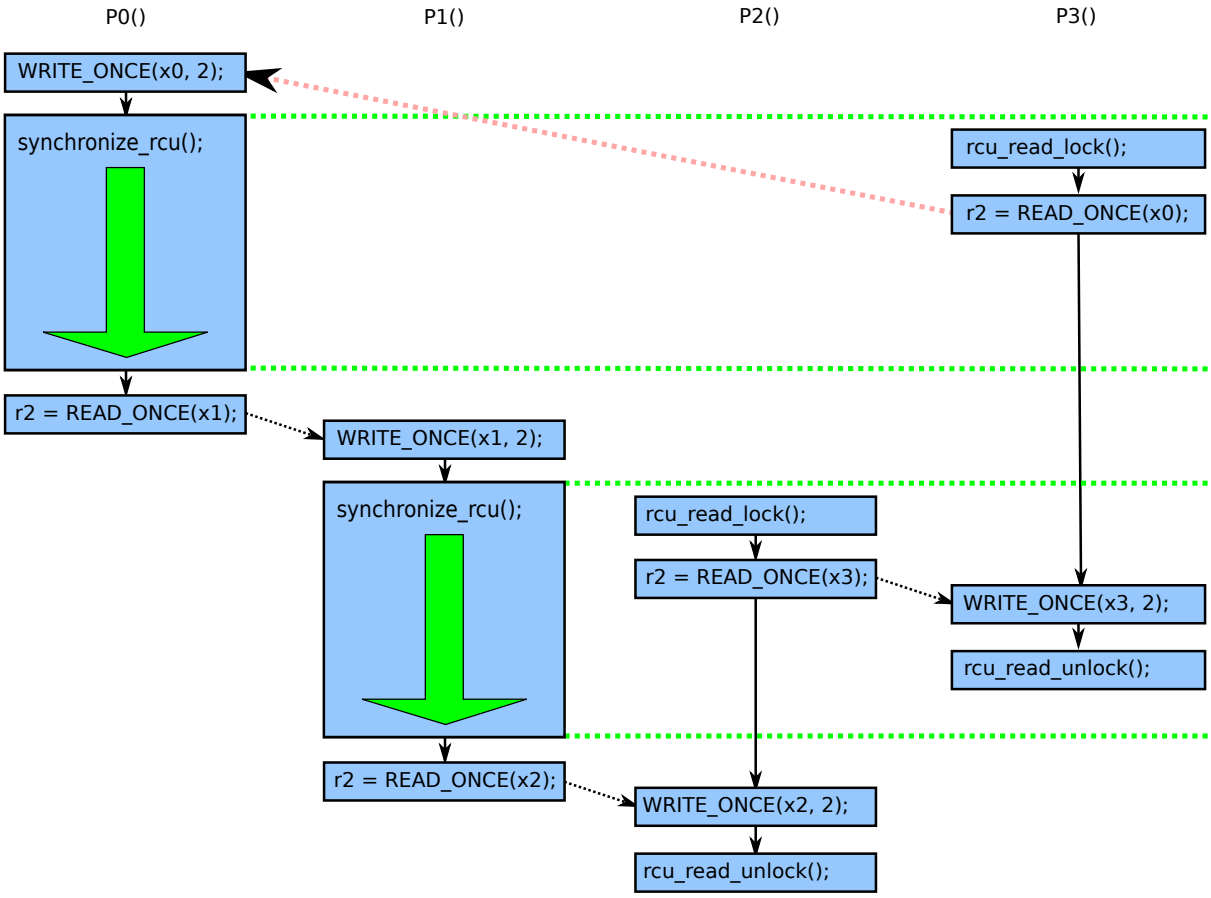


Figure 15.15: No Cycle for Two RCU Grace Periods and Two RCU Readers

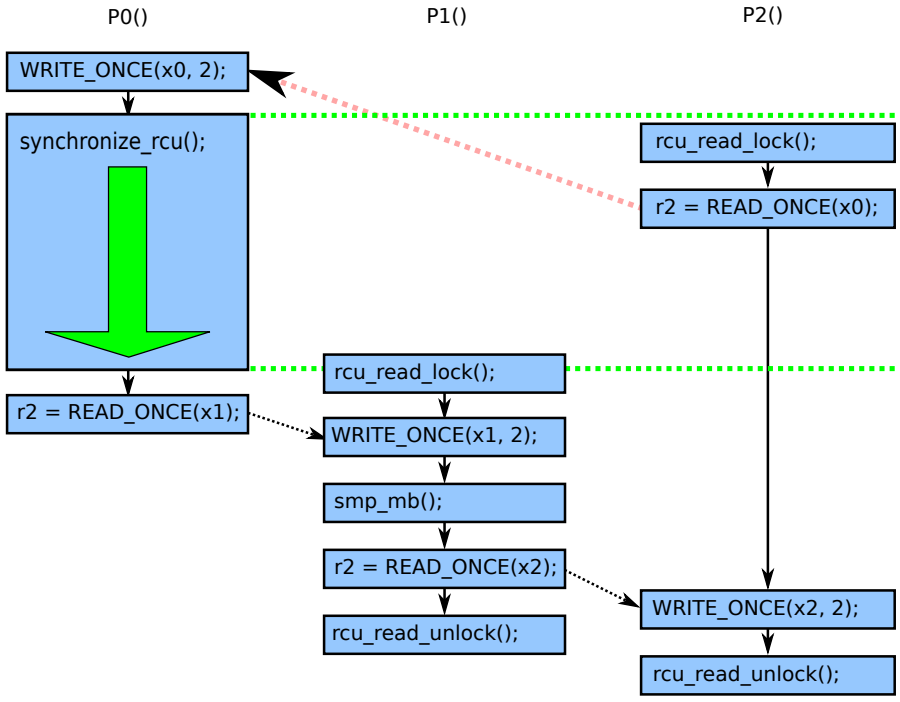


Figure 15.16: Cycle for One RCU Grace Period, Two RCU Readers, and Memory Barrier

with the `smp_mb()` preserving ordering between `P1()` and `P2()`.

Quick Quiz 15.38: What would happen if the `smp_mb()` was instead added between `P2()`'s accesses in Listing 15.40? ■

In short, where RCU's semantics were once purely pragmatic, they are now fully formalized [MW05, DMS+12, GRY13, AMM+18].

It is hoped that detailed semantics for higher-level primitives will enable more capable static analysis and model checking.

15.5 Hardware Specifics

> Rock beats paper!
>
> *Derek Williams*

Each CPU family has its own peculiar approach to memory ordering, which can make portability a challenge, as indicated by Table 15.5. In fact, some software environments simply prohibit direct use of memory-ordering operations, restricting the programmer to mutual-exclusion primitives that incorporate them to the extent that they are required. Please note that this section is not intended to be a reference manual covering all (or even most) aspects of each CPU family, but rather a high-level overview providing a rough comparison. For full details, see the reference manual for the CPU of interest.

Getting back to Table 15.5, the first group of rows look at memory-ordering properties and the second group looks at instruction properties.

The first three rows indicate whether a given CPU allows the four possible combinations of loads and stores to be reordered, as discussed in Section 15.1 and Sections 15.2.2.1–15.2.2.3. The next row ("Atomic Instructions Reordered With Loads or Stores?") indicates whether a given CPU allows loads and stores to be reordered with atomic instructions.

The fifth and sixth rows cover reordering and dependencies, which was covered in Sections 15.2.3–15.2.5 and which is explained in more detail in Section 15.5.1. The short version is that Alpha requires memory barriers for readers as well as updaters of linked data structures, however, these memory barriers are provided by the Alpha architecture-specific code in v4.15 and later Linux kernels.

www.dbooks.org

Table 15.5: Summary of Memory Ordering

		CPU Family								
	Property	Alpha	Armv7-A/R	Armv8	Itanium	MIPS	POWER	SPARC TSO	x86	z Systems
Memory Ordering	Loads Reordered After Loads or Stores?	Y	Y	Y	Y	Y	Y			
	Stores Reordered After Stores?	Y	Y	Y	Y	Y	Y			
	Stores Reordered After Loads?	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Atomic Instructions Reordered With Loads or Stores?	Y	Y	Y		Y	Y			
	Dependent Loads Reordered?	Y								
	Dependent Stores Reordered?									
	Non-Sequentially Consistent?	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Non-Multicopy Atomic?	Y	Y	Y	Y	Y	Y	Y	Y	
	Non-Other-Multicopy Atomic?	Y	Y		Y	Y	Y			
	Non-Cache Coherent?				Y					
Instructions	Load-Acquire/Store-Release?	F	F	i	I	F	b			
	Atomic RMW Instruction Type?	L	L	L	C	L	L	C	C	C
	Incoherent Instruction Cache/Pipeline?	Y	Y	Y	Y	Y	Y	Y	Y	Y

Key: Load-Acquire/Store-Release?
- b: Lightweight memory barrier
- F: Full memory barrier
- i: Instruction with lightweight ordering
- I: Instruction with heavyweight ordering

Atomic RMW Instruction Type?
- C: Compare-and-exchange instruction
- L: Load-linked/store-conditional instruction

The next row, "Non-Sequentially Consistent", indicates whether the CPU's normal load and store instructions are constrained by sequential consistency. Almost all are not constrained in this way for performance reasons.

The next two rows cover multicopy atomicity, which was defined in Section 15.2.7. The first is full-up (and rare) multicopy atomicity, and the second is the weaker other-multicopy atomicity.

The next row, "Non-Cache Coherent", covers accesses from multiple threads to a single variable, which was discussed in Section 15.2.6.

The final three rows cover instruction-level choices and issues. The first row indicates how each CPU implements load-acquire and store-release, the second row classifies CPUs by atomic-instruction type, and the third and final row indicates whether a given CPU has an incoherent instruction cache and pipeline. Such CPUs require special instructions be executed for self-modifying code.

The common "just say no" approach to memory-ordering operations can be eminently reasonable where it applies, but there are environments, such as the Linux kernel, where direct use of memory-ordering operations is required. Therefore, Linux provides a carefully chosen least-common-denominator set of memory-ordering primitives, which are as follows:

`smp_mb()` (full memory barrier) that orders both loads and stores. This means that loads and stores preceding the memory barrier will be committed to memory before any loads and stores following the memory barrier.

`smp_rmb()` (read memory barrier) that orders only loads.

`smp_wmb()` (write memory barrier) that orders only stores.

`smp_mb__before_atomic()` that forces ordering of accesses preceding the `smp_mb__before_atomic()` against accesses following a later RMW atomic operation. This is a noop on systems that fully order atomic RMW operations.

`smp_mb__after_atomic()` that forces ordering of accesses preceding an earlier RMW atomic operation against accesses following the `smp_mb__after_atomic()`. This is also a noop on systems that fully order atomic RMW operations.

`smp_mb__after_spinlock()` that forces ordering of accesses preceding a lock acquisition against accesses following the `smp_mb__after_spinlock()`. This is also a noop on systems that fully order lock acquisitions.

`mmiowb()` that forces ordering on MMIO writes that are guarded by global spinlocks, and is more thoroughly described in a 2016 LWN article on MMIO [MDR16].

The `smp_mb()`, `smp_rmb()`, and `smp_wmb()` primitives also force the compiler to eschew any optimizations that would have the effect of reordering memory optimizations across the barriers.

Quick Quiz 15.39: What happens to code between an atomic operation and an `smp_mb__after_atomic()`? ■

These primitives generate code only in SMP kernels, however, several have UP versions (`mb()`, `rmb()`, and `wmb()`, respectively) that generate a memory barrier even in UP kernels. The `smp_` versions should be used in most cases. However, these latter primitives are useful when writing drivers, because MMIO accesses must remain ordered even in UP kernels. In absence of memory-ordering operations, both CPUs and compilers would happily rearrange these accesses, which at best would make the device act strangely, and could crash your kernel or even damage your hardware.

So most kernel programmers need not worry about the memory-ordering peculiarities of each and every CPU, as long as they stick to these interfaces. If you are working deep in a given CPU's architecture-specific code, of course, all bets are off.

Furthermore, all of Linux's locking primitives (spinlocks, reader-writer locks, semaphores, RCU, . . .) include any needed ordering primitives. So if you are working with code that uses these primitives properly, you need not worry about Linux's memory-ordering primitives.

That said, deep knowledge of each CPU's memory-consistency model can be very helpful when debugging, to say nothing of when writing architecture-specific code or synchronization primitives.

Besides, they say that a little knowledge is a very dangerous thing. Just imagine the damage you could do with a lot of knowledge! For those who wish to understand more about individual CPUs' memory consistency models, the next sections describe those of a few popular and prominent CPUs. Although there is no substitute for actually reading a given CPU's documentation, these sections do give a good overview.

www.dbooks.org

Listing 15.42: Insert and Lock-Free Search (No Ordering)

```
1  struct el *insert(long key, long data)
2  {
3    struct el *p;
4    p = kmalloc(sizeof(*p), GFP_ATOMIC);
5    spin_lock(&mutex);
6    p->next = head.next;
7    p->key = key;
8    p->data = data;
9    smp_store_release(&head.next, p);
10   spin_unlock(&mutex);
11 }
12
13 struct el *search(long searchkey)
14 {
15   struct el *p;
16   p = READ_ONCE_OLD(head.next);
17   while (p != &head) {
18     /* Prior to v4.15, BUG ON ALPHA!!! */
19     if (p->key == searchkey) {
20       return (p);
21     }
22     p = READ_ONCE_OLD(p->next);
23   };
24   return (NULL);
25 }
```

15.5.1 Alpha

It may seem strange to say much of anything about a CPU whose end of life has long since past, but Alpha is interesting because it is the only mainstream CPU that reorders dependent loads, and has thus had outsized influence on concurrency APIs, including within the Linux kernel. The need for core Linux-kernel code to accommodate Alpha ended with version v4.15 of the Linux kernel, and all traces of this accommodation were removed in v5.9 with the removal of the `smp_read_barrier_depends()` and `read_barrier_depends()` APIs. This section is nevertheless retained in the Second Edition because here in early 2021 there are quite a few Linux kernel hackers still working on pre-v4.15 versions of the Linux kernel. In addition, the modifications to `READ_ONCE()` that permitted these APIs to be removed have not necessarily propagated to all userspace projects that might still support Alpha.

The dependent-load difference between Alpha and the other CPUs is illustrated by the code shown in Listing 15.42. This `smp_store_release()` guarantees that the element initialization in lines 6–8 is executed before the element is added to the list on line 9, so that the lock-free search will work correctly. That is, it makes this guarantee on all CPUs *except* Alpha.

Given the pre-v4.15 implementation of `READ_ONCE()`, indicated by `READ_ONCE_OLD()` in the listing, Alpha actually allows the code on line 19 of Listing 15.42 to see the old garbage values that were present before the initialization on lines 6–8.

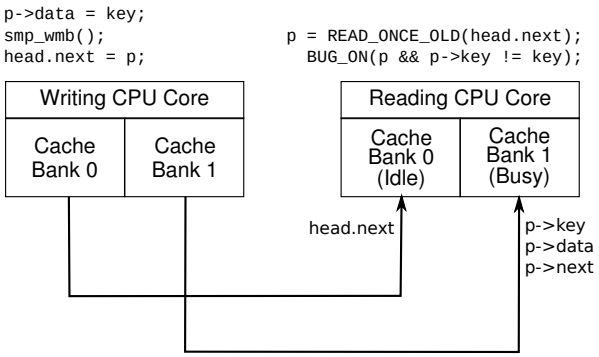


Figure 15.17: Why `smp_read_barrier_depends()` is Required in Pre-v4.15 Linux Kernels

Figure 15.17 shows how this can happen on an aggressively parallel machine with partitioned caches, so that alternating cache lines are processed by the different partitions of the caches. For example, the load of `head.next` on line 16 of Listing 15.42 might access cache bank 0, and the load of `p->key` on line 19 and of `p->next` on line 22 might access cache bank 1. On Alpha, the `smp_store_release()` will guarantee that the cache invalidations performed by lines 6–8 of Listing 15.42 (for `p->next`, `p->key`, and `p->data`) will reach the interconnect before that of line 9 (for `head.next`), but makes absolutely no guarantee about the order of propagation through the reading CPU's cache banks. For example, it is possible that the reading CPU's cache bank 1 is very busy, but cache bank 0 is idle. This could result in the cache invalidations for the new element (`p->next`, `p->key`, and `p->data`) being delayed, so that the reading CPU loads the new value for `head.next`, but loads the old cached values for `p->key` and `p->next`. Yes, this does mean that Alpha can in effect fetch the data pointed to *before* it fetches the pointer itself, strange but true. See the documentation [Com01, Pug00] called out earlier for more information, or if you think that I am just making all this up.[18] The benefit of this unusual approach to ordering is that Alpha can use simpler cache hardware, which in turn permitted higher clock frequencies in Alpha's heyday.

One could place an `smp_rmb()` primitive between the pointer fetch and dereference in order to force Alpha to order the pointer fetch with the later dependent load. However, this imposes unneeded overhead on systems (such as Arm, Itanium, PPC, and SPARC) that respect data

[18] Of course, the astute reader will have already recognized that Alpha is nowhere near as mean and nasty as it could be, the (thankfully) mythical architecture in Appendix C.6.1 being a case in point.

Listing 15.43: Safe Insert and Lock-Free Search

```
1  struct el *insert(long key, long data)
2  {
3    struct el *p;
4    p = kmalloc(sizeof(*p), GFP_ATOMIC);
5    spin_lock(&mutex);
6    p->next = head.next;
7    p->key = key;
8    p->data = data;
9    smp_store_release(&head.next, p);
10   spin_unlock(&mutex);
11 }
12
13 struct el *search(long searchkey)
14 {
15   struct el *p;
16   p = rcu_dereference(head.next);
17   while (p != &head) {
18     if (p->key == searchkey) {
19       return (p);
20     }
21     p = rcu_dereference(p->next);
22   };
23   return (NULL);
24 }
```

dependencies on the read side. A `smp_read_barrier_depends()` primitive was therefore added to the Linux kernel to eliminate overhead on these systems, but was removed in v5.9 of the Linux kernel in favor of augmenting Alpha's definition of `READ_ONCE()`. Thus, as of v5.9, core kernel code no longer needs to concern itself with this aspect of DEC Alpha. However, it is better to use `rcu_dereference()` as shown on lines 16 and 21 of Listing 15.43, which works safely and efficiently for all recent kernel versions.

It is also possible to implement a software mechanism that could be used in place of `smp_store_release()` to force all reading CPUs to see the writing CPU's writes in order. This software barrier could be implemented by sending inter-processor interrupts (IPIs) to all other CPUs. Upon receipt of such an IPI, a CPU would execute a memory-barrier instruction, implementing a system-wide memory barrier similar to that provided by the Linux kernel's `sys_membarrier()` system call. Additional logic is required to avoid deadlocks. Of course, CPUs that respect data dependencies would define such a barrier to simply be `smp_store_release()`. However, this approach was deemed by the Linux community to impose excessive overhead[McK01], and to their point would be completely inappropriate for systems having aggressive real-time response requirements.

The Linux memory-barrier primitives took their names from the Alpha instructions, so `smp_mb()` is `mb`, `smp_rmb()` is `rmb`, and `smp_wmb()` is `wmb`. Alpha is the only CPU whose `READ_ONCE()` includes an `smp_mb()`.

Quick Quiz 15.40: Why does Alpha's `READ_ONCE()` include an `mb()` rather than `rmb()`? ■

Quick Quiz 15.41: Isn't DEC Alpha significant as having the weakest possible memory ordering? ■

For more on Alpha, see its reference manual [Cor02].

15.5.2 Armv7-A/R

The Arm family of CPUs is extremely popular in embedded applications, particularly for power-constrained applications such as cellphones. Its memory model is similar to that of POWER (see Section 15.5.6), but Arm uses a different set of memory-barrier instructions [ARM10]:

`DMB` (data memory barrier) causes the specified type of operations to *appear* to have completed before any subsequent operations of the same type. The "type" of operations can be all operations or can be restricted to only writes (similar to the Alpha `wmb` and the POWER `eieio` instructions). In addition, Arm allows cache coherence to have one of three scopes: single processor, a subset of the processors ("inner") and global ("outer").

`DSB` (data synchronization barrier) causes the specified type of operations to actually complete before any subsequent operations (of any type) are executed. The "type" of operations is the same as that of `DMB`. The `DSB` instruction was called `DWB` (drain write buffer or data write barrier, your choice) in early versions of the Arm architecture.

`ISB` (instruction synchronization barrier) flushes the CPU pipeline, so that all instructions following the `ISB` are fetched only after the `ISB` completes. For example, if you are writing a self-modifying program (such as a JIT), you should execute an `ISB` between generating the code and executing it.

None of these instructions exactly match the semantics of Linux's `rmb()` primitive, which must therefore be implemented as a full `DMB`. The `DMB` and `DSB` instructions have a recursive definition of accesses ordered before and after the barrier, which has an effect similar to that of POWER's cumulativity, both of which are stronger than LKMM's cumulativity described in Section 15.2.7.1.

Arm also implements control dependencies, so that if a conditional branch depends on a load, then any store executed after that conditional branch will be ordered after

www.dbooks.org

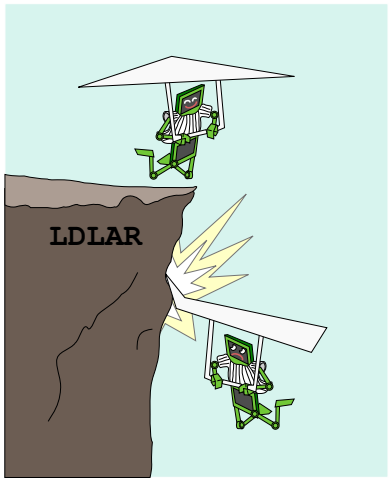


Figure 15.18: Half Memory Barrier

the load. However, loads following the conditional branch will *not* be guaranteed to be ordered unless there is an ISB instruction between the branch and the load. Consider the following example:

```
1 r1 = x;
2 if (r1 == 0)
3   nop();
4 y = 1;
5 r2 = z;
6 ISB();
7 r3 = z;
```

In this example, load-store control dependency ordering causes the load from x on line 1 to be ordered before the store to y on line 4. However, Arm does not respect load-load control dependencies, so that the load on line 1 might well happen *after* the load on line 5. On the other hand, the combination of the conditional branch on line 2 and the ISB instruction on line 6 ensures that the load on line 7 happens after the load on line 1. Note that inserting an additional ISB instruction somewhere between lines 2 and 5 would enforce ordering between lines 1 and 5.

15.5.3 Armv8

Arm's Armv8 CPU family [ARM17] includes 64-bit capabilities, in contrast to their 32-bit-only CPU described in Section 15.5.2. Armv8's memory model closely resembles its Armv7 counterpart, but adds load-acquire (LDLARB, LDLARH, and LDLAR) and store-release (STLLRB, STLLRH, and STLLR) instructions. These instructions act as "half memory barriers", so that Armv8 CPUs can reorder previous accesses with a later LDLAR instruction, but are prohibited from reordering an earlier LDLAR instruction with later accesses, as fancifully depicted in Figure 15.18. Similarly, Armv8 CPUs can reorder an earlier STLLR instruction with a subsequent access, but are prohibited from reordering previous accesses with a later STLLR instruction. As one might expect, this means that these instructions directly support the C11 notion of load-acquire and store-release.

However, Armv8 goes well beyond the C11 memory model by mandating that the combination of a store-release and load-acquire act as a full barrier under certain circumstances. For example, in Armv8, given a store followed by a store-release followed a load-acquire followed by a load, all to different variables and all from a single CPU, all CPUs would agree that the initial store preceded the final load. Interestingly enough, most TSO architectures (including x86 and the mainframe) do not make this guarantee, as the two loads could be reordered before the two stores.

Armv8 is one of only two architectures that needs the smp_mb__after_spinlock() primitive to be a full barrier, due to its relatively weak lock-acquisition implementation in the Linux kernel.

Armv8 also has the distinction of being the first CPU whose vendor publicly defined its memory ordering with an executable formal model [ARM17].

15.5.4 Itanium

Itanium offers a weak consistency model, so that in absence of explicit memory-barrier instructions or dependencies, Itanium is within its rights to arbitrarily reorder memory references [Int02b]. Itanium has a memory-fence instruction named mf, but also has "half-memory fence" modifiers to loads, stores, and to some of its atomic instructions [Int02a]. The acq modifier prevents subsequent memory-reference instructions from being reordered before the acq, but permits prior memory-reference instructions to be reordered after the acq, similar to the Armv8 load-acquire instructions. Similarly, the rel modifier prevents prior memory-reference instructions from being reordered after the rel, but allows subsequent memory-reference instructions to be reordered before the rel.

These half-memory fences are useful for critical sections, since it is safe to push operations into a critical section, but can be fatal to allow them to bleed out. However, as one of the few CPUs with this property, Itanium at one time defined Linux's semantics of memory ordering

associated with lock acquisition and release.[19] Oddly enough, actual Itanium hardware is rumored to implement both load-acquire and store-release instructions as full barriers. Nevertheless, Itanium was the first mainstream CPU to introduce the concept (if not the reality) of load-acquire and store-release into its instruction set.

Quick Quiz 15.42: Given that hardware can have a half memory barrier, why don't locking primitives allow the compiler to move memory-reference instructions into lock-based critical sections? ■

The Itanium `mf` instruction is used for the `smp_rmb()`, `smp_mb()`, and `smp_wmb()` primitives in the Linux kernel. Despite persistent rumors to the contrary, the "`mf`" mnemonic stands for "memory fence".

Itanium also offers a global total order for release operations, including the `mf` instruction. This provides the notion of transitivity, where if a given code fragment sees a given access as having happened, any later code fragment will also see that earlier access as having happened. Assuming, that is, that all the code fragments involved correctly use memory barriers.

Finally, Itanium is the only architecture supporting the Linux kernel that can reorder normal loads to the same variable. The Linux kernel avoids this issue because `READ_ONCE()` emits a `volatile` load, which is compiled as a `ld,acq` instruction, which forces ordering of all `READ_ONCE()` invocations by a given CPU, including those to the same variable.

15.5.5 MIPS

The MIPS memory model [Wav16, page 479] appears to resemble that of Arm, Itanium, and POWER, being weakly ordered by default, but respecting dependencies. MIPS has a wide variety of memory-barrier instructions, but ties them not to hardware considerations, but rather to the use cases provided by the Linux kernel and the C++11 standard [Smi19] in a manner similar to the Armv8 additions:

`SYNC`
: Full barrier for a number of hardware operations in addition to memory references, which is used to implement the v4.13 Linux kernel's `smp_mb()` for OCTEON systems.

`SYNC_WMB`
: Write memory barrier, which can be used on OCTEON systems to implement the `smp_wmb()` primitive in the v4.13 Linux kernel via the `syncw` mnemonic. Other systems use plain `sync`.

`SYNC_MB`
: Full memory barrier, but only for memory operations. This may be used to implement the C++ `atomic_thread_fence(memory_order_seq_cst)`.

`SYNC_ACQUIRE`
: Acquire memory barrier, which could be used to implement C++'s `atomic_thread_fence(memory_order_acquire)`. In theory, it could also be used to implement the v4.13 Linux-kernel `smp_load_acquire()` primitive, but in practice `sync` is used instead.

`SYNC_RELEASE`
: Release memory barrier, which may be used to implement C++'s `atomic_thread_fence(memory_order_release)`. In theory, it could also be used to implement the v4.13 Linux-kernel `smp_store_release()` primitive, but in practice `sync` is used instead.

`SYNC_RMB`
: Read memory barrier, which could in theory be used to implement the `smp_rmb()` primitive in the Linux kernel, except that current MIPS implementations supported by the v4.13 Linux kernel do not need an explicit instruction to force ordering. Therefore, `smp_rmb()` instead simply constrains the compiler.

`SYNCI`
: Instruction-cache synchronization, which is used in conjunction with other instructions to allow self-modifying code, such as that produced by just-in-time (JIT) compilers.

Informal discussions with MIPS architects indicates that MIPS has a definition of transitivity or cumulativity similar to that of Arm and POWER. However, it appears that different MIPS implementations can have different memory-ordering properties, so it is important to consult the documentation for the specific MIPS implementation you are using.

15.5.6 POWER / PowerPC

The POWER and PowerPC CPU families have a wide variety of memory-barrier instructions [IBM94, LHF05]:

[19] PowerPC is now the architecture with this dubious privilege.

www.dbooks.org

`sync` causes all preceding operations to *appear to have* completed before any subsequent operations are started. This instruction is therefore quite expensive.

`lwsync` (lightweight sync) orders loads with respect to subsequent loads and stores, and also orders stores. However, it does *not* order stores with respect to subsequent loads. The `lwsync` instruction may be used to implement load-acquire and store-release operations. Interestingly enough, the `lwsync` instruction enforces the same within-CPU ordering as does x86, z Systems, and coincidentally, SPARC TSO. However, placing the `lwsync` instruction between each pair of memory-reference instructions will *not* result in x86, z Systems, or SPARC TSO memory ordering. On these other systems, if a pair of CPUs independently execute stores to different variables, all other CPUs will agree on the order of these stores. Not so on PowerPC, even with an `lwsync` instruction between each pair of memory-reference instructions, because PowerPC is non-multicopy atomic.

`eieio` (enforce in-order execution of I/O, in case you were wondering) causes all preceding cacheable stores to appear to have completed before all subsequent stores. However, stores to cacheable memory are ordered separately from stores to non-cacheable memory, which means that `eieio` will not force an MMIO store to precede a spinlock release.

`isync` forces all preceding instructions to appear to have completed before any subsequent instructions start execution. This means that the preceding instructions must have progressed far enough that any traps they might generate have either happened or are guaranteed not to happen, and that any side-effects of these instructions (for example, page-table changes) are seen by the subsequent instructions. However, it does *not* force all memory references to be ordered, only the actual execution of the instruction itself. Thus, the loads might return old still-cached values and the stores do not necessarily force flushing of the store buffers.

Unfortunately, none of these instructions line up exactly with Linux's `wmb()` primitive, which requires *all* stores to be ordered, but does not require the other high-overhead actions of the `sync` instruction. But there is no choice: ppc64 versions of `wmb()` and `mb()` are defined to be the heavyweight `sync` instruction. However, Linux's `smp_wmb()` instruction is never used for MMIO (since a driver must carefully order MMIOs in UP as well as SMP kernels, after all), so it is defined to be the lighter weight `eieio` or `lwsync` instruction [MDR16]. This instruction may well be unique in having a five-vowel mnemonic. The `smp_mb()` instruction is also defined to be the `sync` instruction, but both `smp_rmb()` and `rmb()` are defined to be the lighter-weight `lwsync` instruction.

POWER features "cumulativity", which can be used to obtain transitivity. When used properly, any code seeing the results of an earlier code fragment will also see the accesses that this earlier code fragment itself saw. Much more detail is available from McKenney and Silvera [MS09].

POWER respects control dependencies in much the same way that Arm does, with the exception that the POWER `isync` instruction is substituted for the Arm `ISB` instruction.

Like Armv8, POWER requires `smp_mb__after_spinlock()` to be a full memory barrier. In addition, POWER is the only architecture requiring `smp_mb__after_unlock_lock()` to be a full memory barrier. In both cases, this is because of the weak ordering properties of POWER's locking primitives, due to the use of the `lwsync` instruction to provide ordering for both acquisition and release.

Many members of the POWER architecture have incoherent instruction caches, so that a store to memory will not necessarily be reflected in the instruction cache. Thankfully, few people write self-modifying code these days, but JITs and compilers do it all the time. Furthermore, recompiling a recently run program looks just like self-modifying code from the CPU's viewpoint. The `icbi` instruction (instruction cache block invalidate) invalidates a specified cache line from the instruction cache, and may be used in these situations.

15.5.7 SPARC TSO

Although SPARC's TSO (total-store order) is used by both Linux and Solaris, the architecture also defines PSO (partial store order) and RMO (relaxed-memory order). Any program that runs in RMO will also run in either PSO or TSO, and similarly, a program that runs in PSO will also run in TSO. Moving a shared-memory parallel program in the other direction may require careful insertion of memory barriers.

Although SPARC's PSO and RMO modes are not used much these days, they did give rise to a very flexible

memory-barrier instruction [SPA94] that permits fine-grained control of ordering:

`StoreStore` orders preceding stores before subsequent stores. (This option is used by the Linux `smp_wmb()` primitive.)

`LoadStore` orders preceding loads before subsequent stores.

`StoreLoad` orders preceding stores before subsequent loads.

`LoadLoad` orders preceding loads before subsequent loads. (This option is used by the Linux `smp_rmb()` primitive.)

`Sync` fully completes all preceding operations before starting any subsequent operations.

`MemIssue` completes preceding memory operations before subsequent memory operations, important for some instances of memory-mapped I/O.

`Lookaside` does the same as MemIssue, but only applies to preceding stores and subsequent loads, and even then only for stores and loads that access the same memory location.

So, why is "`membar #MemIssue`" needed? Because a "`membar #StoreLoad`" could permit a subsequent load to get its value from a store buffer, which would be disastrous if the write was to an MMIO register that induced side effects on the value to be read. In contrast, "`membar #MemIssue`" would wait until the store buffers were flushed before permitting the loads to execute, thereby ensuring that the load actually gets its value from the MMIO register. Drivers could instead use "`membar #Sync`", but the lighter-weight "`membar #MemIssue`" is preferred in cases where the additional function of the more-expensive "`membar #Sync`" are not required.

The "`membar #Lookaside`" is a lighter-weight version of "`membar #MemIssue`", which is useful when writing to a given MMIO register affects the value that will next be read from that register. However, the heavier-weight "`membar #MemIssue`" must be used when a write to a given MMIO register affects the value that will next be read from *some other* MMIO register.

SPARC requires a `flush` instruction be used between the time that the instruction stream is modified and the time that any of these instructions are executed [SPA94]. This is needed to flush any prior value for that location from the SPARC's instruction cache. Note that `flush` takes an address, and will flush only that address from the instruction cache. On SMP systems, all CPUs' caches are flushed, but there is no convenient way to determine when the off-CPU flushes complete, though there is a reference to an implementation note.

But again, the Linux kernel runs SPARC in TSO mode, so all of the above `membar` variants are strictly of historical interest. In particular, the `smp_mb()` primitive only needs to use `#StoreLoad` because the other three reorderings are prohibited by TSO.

15.5.8 x86

Historically, the x86 CPUs provided "process ordering" so that all CPUs agreed on the order of a given CPU's writes to memory. This allowed the `smp_wmb()` primitive to be a no-op for the CPU [Int04b]. Of course, a compiler directive was also required to prevent optimizations that would reorder across the `smp_wmb()` primitive. In ancient times, certain x86 CPUs gave no ordering guarantees for loads, so the `smp_mb()` and `smp_rmb()` primitives expanded to `lock;addl`. This atomic instruction acts as a barrier to both loads and stores.

But those were ancient times. More recently, Intel has published a memory model for x86 [Int07]. It turns out that Intel's modern CPUs enforce tighter ordering than was claimed in the previous specifications, so this model simply mandates this modern behavior. Even more recently, Intel published an updated memory model for x86 [Int11, Section 8.2], which mandates a total global order for stores, although individual CPUs are still permitted to see their own stores as having happened earlier than this total global order would indicate. This exception to the total ordering is needed to allow important hardware optimizations involving store buffers. In addition, x86 provides other-multicopy atomicity, for example, so that if CPU 0 sees a store by CPU 1, then CPU 0 is guaranteed to see all stores that CPU 1 saw prior to its store. Software may use atomic operations to override these hardware optimizations, which is one reason that atomic operations tend to be more expensive than their non-atomic counterparts.

It is also important to note that atomic instructions operating on a given memory location should all be of the same size [Int16, Section 8.1.2.2]. For example, if you write a program where one CPU atomically increments a byte while another CPU executes a 4-byte atomic increment on that same location, you are on your own.

www.dbooks.org

Some SSE instructions are weakly ordered (`clflush` and non-temporal move instructions [Int04a]). Code that uses these non-temporal move instructions can also use `mfence` for `smp_mb()`, `lfence` for `smp_rmb()`, and `sfence` for `smp_wmb()`. A few older variants of the x86 CPU have a mode bit that enables out-of-order stores, and for these CPUs, `smp_wmb()` must also be defined to be `lock;addl`.

Although newer x86 implementations accommodate self-modifying code without any special instructions, to be fully compatible with past and potential future x86 implementations, a given CPU must execute a jump instruction or a serializing instruction (e.g., `cpuid`) between modifying the code and executing it [Int11, Section 8.1.3].

15.5.9 z Systems

The z Systems machines make up the IBM mainframe family, previously known as the 360, 370, 390 and zSeries [Int04c]. Parallelism came late to z Systems, but given that these mainframes first shipped in the mid 1960s, this is not saying much. The "`bcr 15,0`" instruction is used for the Linux `smp_mb()` primitives, but compiler constraints suffices for both the `smp_rmb()` and `smp_wmb()` primitives. It also has strong memory-ordering semantics, as shown in Table 15.5. In particular, all CPUs will agree on the order of unrelated stores from different CPUs, that is, z Systems is fully multicopy atomic, and is the only commercially available system with this property.

As with most CPUs, the z Systems architecture does not guarantee a cache-coherent instruction stream, hence, self-modifying code must execute a serializing instruction between updating the instructions and executing them. That said, many actual z Systems machines do in fact accommodate self-modifying code without serializing instructions. The z Systems instruction set provides a large set of serializing instructions, including compare-and-swap, some types of branches (for example, the aforementioned "`bcr 15,0`" instruction), and test-and-set.

15.6 Where is Memory Ordering Needed?

> Almost all people are intelligent. It is method that they lack.
>
> *F. W. Nichol*

This section revisits Table 15.3 and Section 15.1.3, summarizing the intervening discussion with a more sophisticated set of rules of thumb.

The first rule of thumb is that memory-ordering operations are only required where there is a possibility of interaction between at least two variables shared among at least two threads. In light of the intervening material, this single sentence encapsulates much of Section 15.1.3's basic rules of thumb, for example, keeping in mind that "memory-barrier pairing" is a two-thread special case of "cycle". And, as always, if a single-threaded program will provide sufficient performance, why bother with parallelism?[20] After all, avoiding parallelism also avoids the added cost of memory-ordering operations.

The second rule of thumb involves load-buffering situations: If all thread-to-thread communication in a given cycle use store-to-load links (that is, the next thread's load returns the value stored by the previous thread), minimal ordering suffices. Minimal ordering includes dependencies and acquires as well as all stronger ordering operations.

The third rule of thumb involves release-acquire chains: If all but one of the links in a given cycle is a store-to-load link, it is sufficient to use release-acquire pairs for each of those store-to-load links, as illustrated by Listings 15.23 and 15.24. You can replace a given acquire with a dependency in environments permitting this, keeping in mind that the C11 standard's memory model does *not* fully respect dependencies. Therefore, a dependency leading to a load must be headed by a `READ_ONCE()` or an `rcu_dereference()`: a plain C-language load is not sufficient. In addition, carefully review Sections 15.3.2 and 15.3.3, because a dependency broken by your compiler will not order anything. The two threads sharing the sole non-store-to-load link can usually substitute `WRITE_ONCE()` plus `smp_wmb()` for `smp_store_release()` on the one hand, and `READ_ONCE()` plus `smp_rmb()` for `smp_load_acquire()` on the other. However, the wise developer

[20] Hobbyists and researchers should of course feel free to ignore this and many other cautions.

will check such substitutions carefully, for example, using the herd tool as described in Section 12.3.

Quick Quiz 15.43: Why is it necessary to use heavier-weight ordering for load-to-store and store-to-store links, but not for store-to-load links? What on earth makes store-to-load links so special??? ■

The fourth and final rule of thumb identifies where full memory barriers (or stronger) are required: If a given cycle contains two or more non-store-to-load links (that is, a total of two or more links that are either load-to-store or store-to-store links), you will need at least one full barrier between each pair of non-store-to-load links in that cycle, as illustrated by Listing 15.19 as well as in the answer to Quick Quiz 15.24. Full barriers include `smp_mb()`, successful full-strength non-`void` atomic RMW operations, and other atomic RMW operations in conjunction with either `smp_mb__before_atomic()` or `smp_mb__after_atomic()`. Any of RCU's grace-period-wait primitives (`synchronize_rcu()` and friends) also act as full barriers, but at far greater expense than `smp_mb()`. With strength comes expense, though full barriers usually hurt performance more than they hurt scalability.

Recapping the rules:

1. Memory-ordering operations are required only if at least two variables are shared by at least two threads.
2. If all links in a cycle are store-to-load links, then minimal ordering suffices.
3. If all but one of the links in a cycle are store-to-load links, then each store-to-load link may use a release-acquire pair.
4. Otherwise, at least one full barrier is required between each pair of non-store-to-load links.

Note that these four rules of thumb encapsulate *minimum* guarantees. A given architecture may give more substantial guarantees, as discussed in Section 15.5, but these guarantees may only be relied upon in code that runs only for that architecture. In addition, more involved memory models may give stronger guarantees [AMM+18], at the expense of somewhat greater complexity.

One final word of advice: Use of raw memory-ordering primitives is a last resort. It is almost always better to use existing primitives, such as locking or RCU, thus letting those primitives do the memory ordering for you.

www.dbooks.org

Creating a perfect API is like committing the perfect crime. There are at least fifty things that can go wrong, and if you are a genius, you might be able to anticipate twenty-five of them.

With apologies to any Kathleen Turner fans who might still be alive.

Chapter 16

Ease of Use

16.1 What is Easy?

When someone says "I want a programming language in which I need only say what I wish done," give them a lollipop.

Alan J. Perlis, updated

If you are tempted to look down on ease-of-use requirements, please consider that an ease-of-use bug in Linux-kernel RCU resulted in an exploitable Linux-kernel security bug in a use of RCU [McK19a]. It is therefore clearly important that even in-kernel APIs be easy to use.

Unfortunately, "easy" is a relative term. For example, many people would consider a 15-hour airplane flight to be a bit of an ordeal—unless they stopped to consider alternative modes of transportation, especially swimming. This means that creating an easy-to-use API requires that you understand your intended users well enough to know what is easy for them. Which might or might not have anything to do with what is easy for you.

The following question illustrates this point: "Given a randomly chosen person among everyone alive today, what one change would improve that person's life?"

There is no single change that would be guaranteed to help everyone's life. After all, there is an extremely wide range of people, with a correspondingly wide range of needs, wants, desires, and aspirations. A starving person might need food, but additional food might well hasten the death of a morbidly obese person. The high level of excitement so fervently desired by many young people might well be fatal to someone recovering from a heart attack. Information critical to the success of one person might contribute to the failure of someone suffering from information overload. In short, if you are working on a software project that is intended to help people you know nothing about, you should not be surprised when those people find fault with your project.

If you really want to help a given group of people, there is simply no substitute for working closely with them over an extended period of time, as in years. Nevertheless, there are some simple things that you can do to increase the odds of your users being happy with your software, and some of these things are covered in the next section.

16.2 Rusty Scale for API Design

Finding the appropriate measurement is thus not a mathematical exercise. It is a risk-taking judgment.

Peter Drucker

This section is adapted from portions of Rusty Russell's 2003 Ottawa Linux Symposium keynote address [Rus03, Slides 39–57]. Rusty's key point is that the goal should not be merely to make an API easy to use, but rather to make the API hard to misuse. To that end, Rusty proposed his "Rusty Scale" in decreasing order of this important hard-to-misuse property.

The following list attempts to generalize the Rusty Scale beyond the Linux kernel:

1. It is impossible to get wrong. Although this is the standard to which all API designers should strive, only the mythical `dwim()`[1] command manages to come close.
2. The compiler or linker won't let you get it wrong.
3. The compiler or linker will warn you if you get it wrong. `BUILD_BUG_ON()` is your users' friend.

[1] The `dwim()` function is an acronym that expands to "do what I mean".

Edition.2-rc11

www.dbooks.org

4. The simplest use is the correct one.

5. The name tells you how to use it. But names can be two-edged swords. Although `rcu_read_lock()` is plain enough for someone converting code from reader-writer locking, it might cause some consternation for someone converting code from reference counting.

6. Do it right or it will always break at runtime. `WARN_ON_ONCE()` is your users' friend.

7. Follow common convention and you will get it right. The `malloc()` library function is a good example. Although it is easy to get memory allocation wrong, a great many projects do manage to get it right, at least most of the time. Using `malloc()` in conjunction with Valgrind [The11] moves `malloc()` almost up to the "do it right or it will always break at runtime" point on the scale.

8. Read the documentation and you will get it right.

9. Read the implementation and you will get it right.

10. Read the right mailing-list archive and you will get it right.

11. Read the right mailing-list archive and you will get it wrong.

12. Read the implementation and you will get it wrong. The original non-`CONFIG_PREEMPT` implementation of `rcu_read_lock()` [McK07a] is an infamous example of this point on the scale.

13. Read the documentation and you will get it wrong. For example, the DEC Alpha `wmb` instruction's documentation [Cor02] fooled a number of developers into thinking that this instruction had much stronger memory-order semantics than it actually does. Later documentation clarified this point [Com01, Pug00], moving the `wmb` instruction up to the "read the documentation and you will get it right" point on the scale.

14. Follow common convention and you will get it wrong. The `printf()` statement is an example of this point on the scale because developers almost always fail to check `printf()`'s error return.

15. Do it right and it will break at runtime.

16. The name tells you how not to use it.

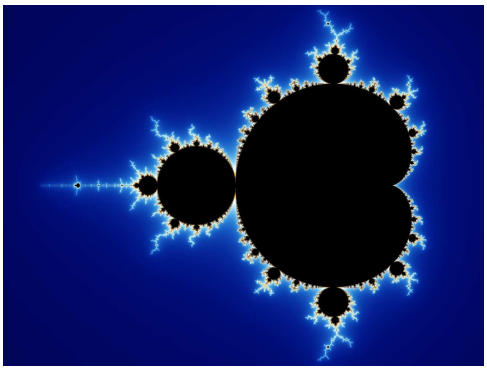


Figure 16.1: Mandelbrot Set (Courtesy of Wikipedia)

17. The obvious use is wrong. The Linux kernel `smp_mb()` function is an example of this point on the scale. Many developers assume that this function has much stronger ordering semantics than it actually possesses. Chapter 15 contains the information needed to avoid this mistake, as does the Linux-kernel source tree's `Documentation` and `tools/memory-model` directories.

18. The compiler or linker will warn you if you get it right.

19. The compiler or linker won't let you get it right.

20. It is impossible to get right. The `gets()` function is a famous example of this point on the scale. In fact, `gets()` can perhaps best be described as an unconditional buffer-overflow security hole.

16.3 Shaving the Mandelbrot Set

> Simplicity does not precede complexity,
> but follows it.
>
> *Alan J. Perlis*

The set of useful programs resembles the Mandelbrot set (shown in Figure 16.1) in that it does not have a clear-cut smooth boundary—if it did, the halting problem would be solvable. But we need APIs that real people can use, not ones that require a Ph.D. dissertation be completed for each and every potential use. So, we "shave the Mandelbrot set",[2] restricting the use of the API to an easily described subset of the full set of potential uses.

[2] Due to Josh Triplett.

Such shaving may seem counterproductive. After all, if an algorithm works, why shouldn't it be used?

To see why at least some shaving is absolutely necessary, consider a locking design that avoids deadlock, but in perhaps the worst possible way. This design uses a circular doubly linked list, which contains one element for each thread in the system along with a header element. When a new thread is spawned, the parent thread must insert a new element into this list, which requires some sort of synchronization.

One way to protect the list is to use a global lock. However, this might be a bottleneck if threads were being created and deleted frequently.[3] Another approach would be to use a hash table and to lock the individual hash buckets, but this can perform poorly when scanning the list in order.

A third approach is to lock the individual list elements, and to require the locks for both the predecessor and successor to be held during the insertion. Since both locks must be acquired, we need to decide which order to acquire them in. Two conventional approaches would be to acquire the locks in address order, or to acquire them in the order that they appear in the list, so that the header is always acquired first when it is one of the two elements being locked. However, both of these methods require special checks and branches.

The to-be-shaven solution is to unconditionally acquire the locks in list order. But what about deadlock?

Deadlock cannot occur.

To see this, number the elements in the list starting with zero for the header up to N for the last element in the list (the one preceding the header, given that the list is circular). Similarly, number the threads from zero to $N-1$. If each thread attempts to lock some consecutive pair of elements, at least one of the threads is guaranteed to be able to acquire both locks.

Why?

Because there are not enough threads to reach all the way around the list. Suppose thread 0 acquires element 0's lock. To be blocked, some other thread must have already acquired element 1's lock, so let us assume that thread 1 has done so. Similarly, for thread 1 to be blocked, some other thread must have acquired element 2's lock, and so on, up through thread $N-1$, who acquires element $N-1$'s lock. For thread $N-1$ to be blocked, some other thread must have acquired element N's lock. But there are no more threads, and so thread $N-1$ cannot be blocked. Therefore, deadlock cannot occur.

[3] Those of you with strong operating-system backgrounds, please suspend disbelief. Those unable to suspend disbelief are encouraged to provide better examples.

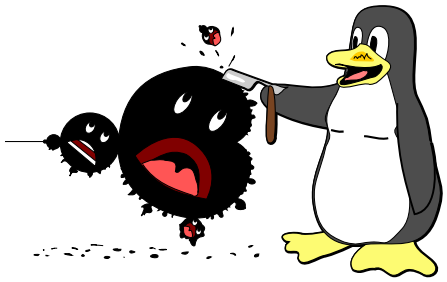


Figure 16.2: Shaving the Mandelbrot Set

So why should we prohibit use of this delightful little algorithm?

The fact is that if you *really* want to use it, we cannot stop you. We *can*, however, recommend against such code being included in any project that we care about.

But, before you use this algorithm, please think through the following Quick Quiz.

Quick Quiz 16.1: Can a similar algorithm be used when deleting elements? ■

The fact is that this algorithm is extremely specialized (it only works on certain sized lists), and also quite fragile. Any bug that accidentally failed to add a node to the list could result in deadlock. In fact, simply adding the node a bit too late could result in deadlock, as could increasing the number of threads.

In addition, the other algorithms described above are "good and sufficient". For example, simply acquiring the locks in address order is fairly simple and quick, while allowing the use of lists of any size. Just be careful of the special cases presented by empty lists and lists containing only one element!

Quick Quiz 16.2: Yetch! What ever possessed someone to come up with an algorithm that deserves to be shaved as much as this one does??? ■

In summary, we do not use algorithms simply because they happen to work. We instead restrict ourselves to algorithms that are useful enough to make it worthwhile learning about them. The more difficult and complex the algorithm, the more generally useful it must be in order for the pain of learning it and fixing its bugs to be worthwhile.

Quick Quiz 16.3: Give an exception to this rule. ■

www.dbooks.org

Exceptions aside, we must continue to shave the software "Mandelbrot set" so that our programs remain maintainable, as shown in Figure 16.2.

Prediction is very difficult, especially about the future.

Niels Bohr

Chapter 17

Conflicting Visions of the Future

This chapter presents some conflicting visions of the future of parallel programming. It is not clear which of these will come to pass, in fact, it is not clear that any of them will. They are nevertheless important because each vision has its devoted adherents, and if enough people believe in something fervently enough, you will need to deal with that thing's existence in the form of its influence on the thoughts, words, and deeds of its adherents. Besides which, one or more of these visions will actually come to pass. But most are bogus. Tell which is which and you'll be rich [Spi77]!

Therefore, the following sections give an overview of transactional memory, hardware transactional memory, formal verification in regression testing, and parallel functional programming. But first, a cautionary tale on prognostication taken from the early 2000s.

17.1 The Future of CPU Technology Ain't What it Used to Be

A great future behind him.

David Maraniss

Years past always seem so simple and innocent when viewed through the lens of many years of experience. And the early 2000s were for the most part innocent of the impending failure of Moore's Law to continue delivering the then-traditional increases in CPU clock frequency. Oh, there were the occasional warnings about the limits of technology, but such warnings had been sounded for decades. With that in mind, consider the following scenarios:

1. Uniprocessor Über Alles (Figure 17.1),
2. Multithreaded Mania (Figure 17.2),
3. More of the Same (Figure 17.3), and
4. Crash Dummies Slamming into the Memory Wall (Figure 17.4).
5. Astounding Accelerators (Figure 17.5).

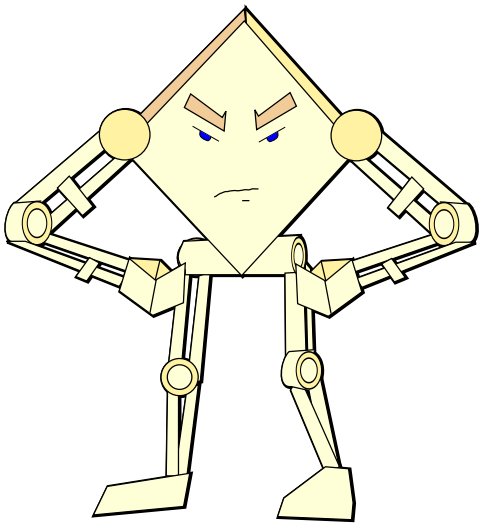


Figure 17.1: Uniprocessor Über Alles

Each of these scenarios is covered in the following sections.

17.1.1 Uniprocessor Über Alles

As was said in 2004 [McK04]:

www.dbooks.org

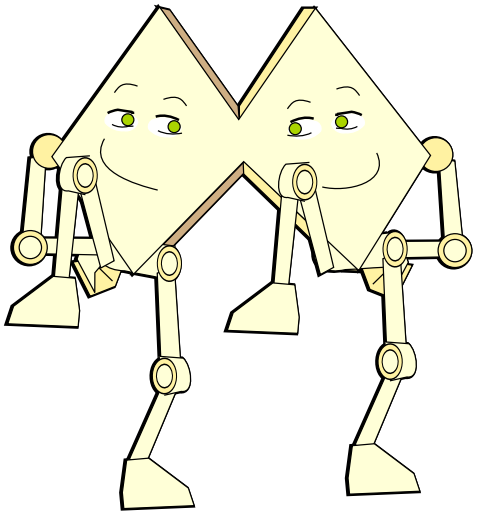


Figure 17.2: Multithreaded Mania

> In this scenario, the combination of Moore's-Law increases in CPU clock rate and continued progress in horizontally scaled computing render SMP systems irrelevant. This scenario is therefore dubbed "Uniprocessor Über Alles", literally, uniprocessors above all else.
>
> These uniprocessor systems would be subject only to instruction overhead, since memory barriers, cache thrashing, and contention do not affect single-CPU systems. In this scenario, RCU is useful only for niche applications, such as interacting with NMIs. It is not clear that an operating system lacking RCU would see the need to adopt it, although operating systems that already implement RCU might continue to do so.
>
> However, recent progress with multithreaded CPUs seems to indicate that this scenario is quite unlikely.

Unlikely indeed! But the larger software community was reluctant to accept the fact that they would need to embrace parallelism, and so it was some time before this community concluded that the "free lunch" of Moore's-Law-induced CPU core-clock frequency increases was well and truly finished. Never forget: belief is an emotion, not necessarily the result of a rational technical thought process!

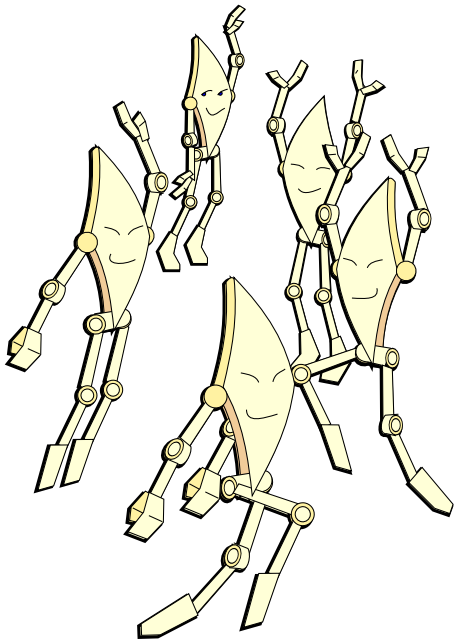


Figure 17.3: More of the Same

17.1.2 Multithreaded Mania

Also from 2004 [McK04]:

> A less-extreme variant of Uniprocessor Über Alles features uniprocessors with hardware multithreading, and in fact multithreaded CPUs are now standard for many desktop and laptop computer systems. The most aggressively multithreaded CPUs share all levels of cache hierarchy, thereby eliminating CPU-to-CPU memory latency, in turn greatly reducing the performance penalty for traditional synchronization mechanisms. However, a multithreaded CPU would still incur overhead due to contention and to pipeline stalls caused by memory barriers. Furthermore, because all hardware threads share all levels of cache, the cache available to a given hardware thread is a fraction of what it would be on an equivalent single-threaded CPU, which can degrade performance for applications with large cache footprints. There is also some possibility that the restricted amount of cache available will cause RCU-based algorithms to incur performance penalties due to their grace-period-induced additional memory

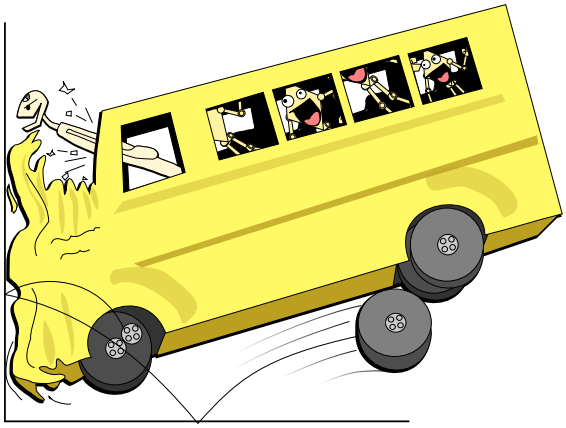


Figure 17.4: Crash Dummies Slamming into the Memory Wall

> consumption. Investigating this possibility is future work.
>
> However, in order to avoid such performance degradation, a number of multithreaded CPUs and multi-CPU chips partition at least some of the levels of cache on a per-hardware-thread basis. This increases the amount of cache available to each hardware thread, but re-introduces memory latency for cachelines that are passed from one hardware thread to another.

And we all know how this story has played out, with multiple multi-threaded cores on a single die plugged into a single socket, with varying degrees of optimization for lower numbers of active threads per core. The question then becomes whether or not future shared-memory systems will always fit into a single socket.

17.1.3 More of the Same

Again from 2004 [McK04]:

> The More-of-the-Same scenario assumes that the memory-latency ratios will remain roughly where they are today.
>
> This scenario actually represents a change, since to have more of the same, interconnect performance must begin keeping up with the Moore's-Law increases in core CPU performance. In this scenario, overhead due to pipeline stalls, memory latency, and contention remains significant,

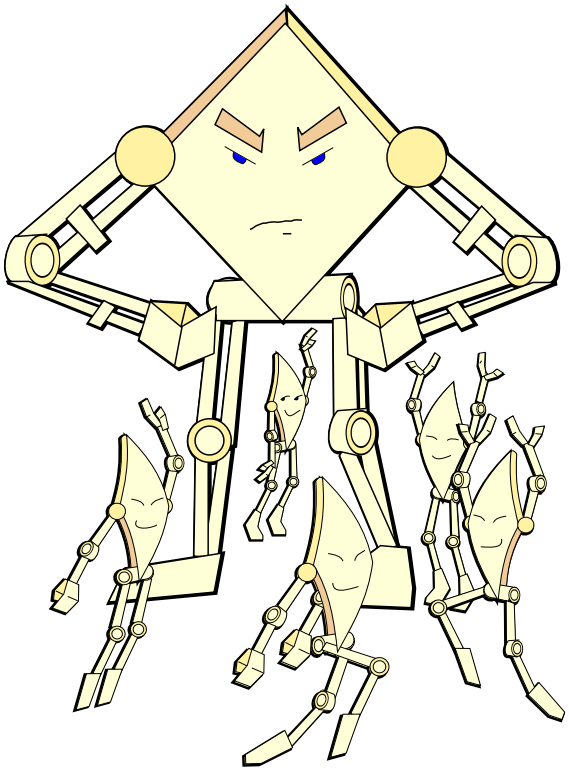


Figure 17.5: Astounding Accelerators

> and RCU retains the high level of applicability that it enjoys today.

And the change has been the ever-increasing levels of integration that Moore's Law is still providing. But longer term, which will it be? More CPUs per die? Or more I/O, cache, and memory?

Servers seem to be choosing the former, while embedded systems on a chip (SoCs) continue choosing the latter.

17.1.4 Crash Dummies Slamming into the Memory Wall

And one more quote from 2004 [McK04]:

> If the memory-latency trends shown in Figure 17.6 continue, then memory latency will continue to grow relative to instruction-execution overhead. Systems such as Linux that have significant use of RCU will find additional use of RCU to be profitable, as shown in Figure 17.7.

www.dbooks.org

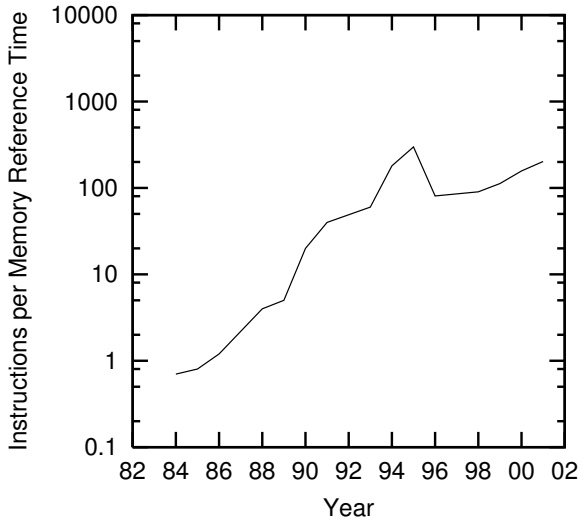


Figure 17.6: Instructions per Local Memory Reference for Sequent Computers

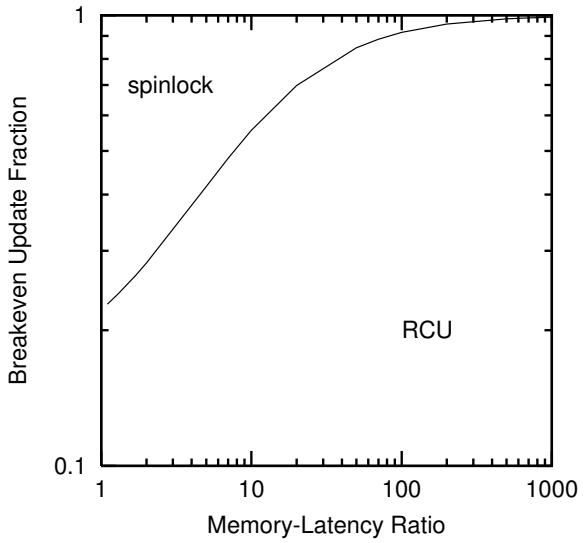


Figure 17.7: Breakevens vs. r, λ Large, Four CPUs

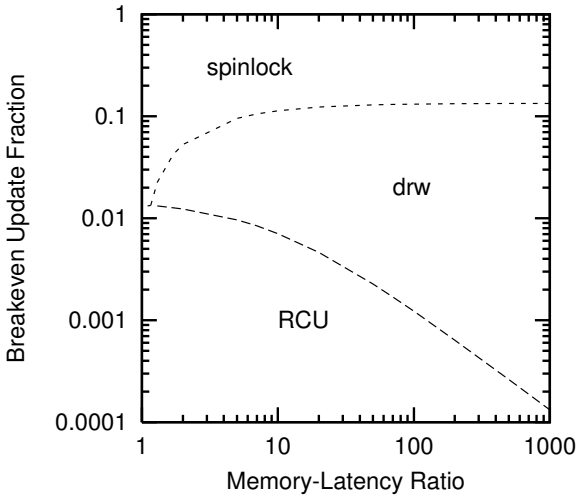


Figure 17.8: Breakevens vs. r, λ Small, Four CPUs

> As can be seen in this figure, if RCU is heavily used, increasing memory-latency ratios give RCU an increasing advantage over other synchronization mechanisms. In contrast, systems with minor use of RCU will require increasingly high degrees of read intensity for use of RCU to pay off, as shown in Figure 17.8. As can be seen in this figure, if RCU is lightly used, increasing memory-latency ratios put RCU at an increasing disadvantage compared to other synchronization mechanisms. Since Linux has been observed with over 1,600 callbacks per grace period under heavy load [SM04b], it seems safe to say that Linux falls into the former category.

On the one hand, this passage failed to anticipate the cache-warmth issues that RCU can suffer from in workloads with significant update intensity, in part because it seemed unlikely that RCU would really be used for such workloads. In the event, the `SLAB_TYPESAFE_BY_RCU` has been pressed into service in a number of instances where these cache-warmth issues would otherwise be problematic, as has sequence locking. On the other hand, this passage also failed to anticipate that RCU would be used to reduce scheduling latency or for security.

Much of the data generated for this book was collected on an eight-socket system with 28 cores per socket and two hardware threads per core, for a total of 448 hardware threads. The idle-system memory latencies are less than one microsecond, which are no worse than those of similar-sized systems of the year 2004. Some claim that these

latencies approach a microsecond only because of the x86 CPU family's relatively strong memory ordering, but it may be some time before that particular argument is settled.

17.1.5 Astounding Accelerators

The potential of hardware accelerators was not quite as clear in 2004 as it is in 2021, so this section has no quote. However, the November 2020 Top 500 list [MDSS20] features a great many accelerators, so one could argue that this section is a view of the present rather than of the future. The same could be said of most of the preceding sections.

Hardware accelerators are being put to many other uses, including encryption, compression, machine learning.

In short, beware of prognostications, including those in the remainder of this chapter.

17.2 Transactional Memory

> Everything should be as simple as it can be, but not simpler.
>
> *Albert Einstein, by way of Louis Zukofsky*

The idea of using transactions outside of databases goes back many decades [Lom77, Kni86, HM93], with the key difference between database and non-database transactions being that non-database transactions drop the "D" in the "ACID"[1] properties defining database transactions. The idea of supporting memory-based transactions, or "transactional memory" (TM), in hardware is more recent [HM93], but unfortunately, support for such transactions in commodity hardware was not immediately forthcoming, despite other somewhat similar proposals being put forward [SSHT93]. Not long after, Shavit and Touitou proposed a software-only implementation of transactional memory (STM) that was capable of running on commodity hardware, give or take memory-ordering issues [ST95]. This proposal languished for many years, perhaps due to the fact that the research community's attention was absorbed by non-blocking synchronization (see Section 14.2).

But by the turn of the century, TM started receiving more attention [MT01, RG01], and by the middle of the decade, the level of interest can only be termed "incandescent" [Her05, Gro07], with only a few voices of caution [BLM05, MMW07].

The basic idea behind TM is to execute a section of code atomically, so that other threads see no intermediate state. As such, the semantics of TM could be implemented by simply replacing each transaction with a recursively acquirable global lock acquisition and release, albeit with abysmal performance and scalability. Much of the complexity inherent in TM implementations, whether hardware or software, is efficiently detecting when concurrent transactions can safely run in parallel. Because this detection is done dynamically, conflicting transactions can be aborted or "rolled back", and in some implementations, this failure mode is visible to the programmer.

Because transaction roll-back is increasingly unlikely as transaction size decreases, TM might become quite attractive for small memory-based operations, such as linked-list manipulations used for stacks, queues, hash tables, and search trees. However, it is currently much more difficult to make the case for large transactions, particularly those containing non-memory operations such as I/O and process creation. The following sections look at current challenges to the grand vision of "Transactional Memory Everywhere" [McK09b]. Section 17.2.1 examines the challenges faced interacting with the outside world, Section 17.2.2 looks at interactions with process modification primitives, Section 17.2.3 explores interactions with other synchronization primitives, and finally Section 17.2.4 closes with some discussion.

17.2.1 Outside World

In the wise words of Donald Knuth:

> Many computer users feel that input and output are not actually part of "real programming," they are merely things that (unfortunately) must be done in order to get information in and out of the machine.

Whether we believe that input and output are "real programming" or not, the fact is that for most computer systems, interaction with the outside world is a first-class requirement. This section therefore critiques transactional memory's ability to so interact, whether via I/O operations, time delays, or persistent storage.

[1] Atomicity, consistency, isolation, and durability.

www.dbooks.org

17.2.1.1 I/O Operations

One can execute I/O operations within a lock-based critical section, while holding a hazard pointer, within a sequence-locking read-side critical section, and from within a userspace-RCU read-side critical section, and even all at the same time, if need be. What happens when you attempt to execute an I/O operation from within a transaction?

The underlying problem is that transactions may be rolled back, for example, due to conflicts. Roughly speaking, this requires that all operations within any given transaction be revocable, so that executing the operation twice has the same effect as executing it once. Unfortunately, I/O is in general the prototypical irrevocable operation, making it difficult to include general I/O operations in transactions. In fact, general I/O is irrevocable: Once you have pushed the proverbial button launching the nuclear warheads, there is no turning back.

Here are some options for handling of I/O within transactions:

1. Restrict I/O within transactions to buffered I/O with in-memory buffers. These buffers may then be included in the transaction in the same way that any other memory location might be included. This seems to be the mechanism of choice, and it does work well in many common cases of situations such as stream I/O and mass-storage I/O. However, special handling is required in cases where multiple record-oriented output streams are merged onto a single file from multiple processes, as might be done using the "a+" option to `fopen()` or the `O_APPEND` flag to `open()`. In addition, as will be seen in the next section, common networking operations cannot be handled via buffering.

2. Prohibit I/O within transactions, so that any attempt to execute an I/O operation aborts the enclosing transaction (and perhaps multiple nested transactions). This approach seems to be the conventional TM approach for unbuffered I/O, but requires that TM interoperate with other synchronization primitives tolerating I/O.

3. Prohibit I/O within transactions, but enlist the compiler's aid in enforcing this prohibition.

4. Permit only one special *irrevocable* transaction [SMS08] to proceed at any given time, thus allowing irrevocable transactions to contain I/O operations.[2] This works in general, but severely limits the scalability and performance of I/O operations. Given that scalability and performance is a first-class goal of parallelism, this approach's generality seems a bit self-limiting. Worse yet, use of irrevocability to tolerate I/O operations seems to greatly restrict use of manual transaction-abort operations.[3] Finally, if there is an irrevocable transaction manipulating a given data item, any other transaction manipulating that same data item cannot have non-blocking semantics.

5. Create new hardware and protocols such that I/O operations can be pulled into the transactional substrate. In the case of input operations, the hardware would need to correctly predict the result of the operation, and to abort the transaction if the prediction failed.

I/O operations are a well-known weakness of TM, and it is not clear that the problem of supporting I/O in transactions has a reasonable general solution, at least if "reasonable" is to include usable performance and scalability. Nevertheless, continued time and attention to this problem will likely produce additional progress.

17.2.1.2 RPC Operations

One can execute RPCs within a lock-based critical section, while holding a hazard pointer, within a sequence-locking read-side critical section, and from within a userspace-RCU read-side critical section, and even all at the same time, if need be. What happens when you attempt to execute an RPC from within a transaction?

If both the RPC request and its response are to be contained within the transaction, and if some part of the transaction depends on the result returned by the response, then it is not possible to use the memory-buffer tricks that can be used in the case of buffered I/O. Any attempt to take this buffering approach would deadlock the transaction, as the request could not be transmitted until the transaction was guaranteed to succeed, but the transaction's success might not be knowable until after the response is received, as is the case in the following example:

```
1 begin_trans();
2 rpc_request();
3 i = rpc_response();
```

[2] In earlier literature, irrevocable transactions are termed *inevitable* transactions.

[3] This difficulty was pointed out by Michael Factor. To see the problem, think through what TM should do in response to an attempt to abort a transaction after it has executed an irrevocable operation.

```
4 a[i]++;
5 end_trans();
```

The transaction's memory footprint cannot be determined until after the RPC response is received, and until the transaction's memory footprint can be determined, it is impossible to determine whether the transaction can be allowed to commit. The only action consistent with transactional semantics is therefore to unconditionally abort the transaction, which is, to say the least, unhelpful.

Here are some options available to TM:

1. Prohibit RPC within transactions, so that any attempt to execute an RPC operation aborts the enclosing transaction (and perhaps multiple nested transactions). Alternatively, enlist the compiler to enforce RPC-free transactions. This approach does work, but will require TM to interact with other synchronization primitives.

2. Permit only one special irrevocable transaction [SMS08] to proceed at any given time, thus allowing irrevocable transactions to contain RPC operations. This works in general, but severely limits the scalability and performance of RPC operations. Given that scalability and performance is a first-class goal of parallelism, this approach's generality seems a bit self-limiting. Furthermore, use of irrevocable transactions to permit RPC operations restricts manual transaction-abort operations once the RPC operation has started. Finally, if there is an irrevocable transaction manipulating a given data item, any other transaction manipulating that same data item must have blocking semantics.

3. Identify special cases where the success of the transaction may be determined before the RPC response is received, and automatically convert these to irrevocable transactions immediately before sending the RPC request. Of course, if several concurrent transactions attempt RPC calls in this manner, it might be necessary to roll all but one of them back, with consequent degradation of performance and scalability. This approach nevertheless might be valuable given long-running transactions ending with an RPC. This approach must still restrict manual transaction-abort operations.

4. Identify special cases where the RPC response may be moved out of the transaction, and then proceed using techniques similar to those used for buffered I/O.

5. Extend the transactional substrate to include the RPC server as well as its client. This is in theory possible, as has been demonstrated by distributed databases. However, it is unclear whether the requisite performance and scalability requirements can be met by distributed-database techniques, given that memory-based TM has no slow disk drives behind which to hide such latencies. Of course, given the advent of solid-state disks, it is also quite possible that databases will need to redesign their approach to latency hiding.

As noted in the prior section, I/O is a known weakness of TM, and RPC is simply an especially problematic case of I/O.

17.2.1.3 Time Delays

An important special case of interaction with extra-transactional accesses involves explicit time delays within a transaction. Of course, the idea of a time delay within a transaction flies in the face of TM's atomicity property, but this sort of thing is arguably what weak atomicity is all about. Furthermore, correct interaction with memory-mapped I/O sometimes requires carefully controlled timing, and applications often use time delays for varied purposes. Finally, one can execute time delays within a lock-based critical section, while holding a hazard pointer, within a sequence-locking read-side critical section, and from within a userspace-RCU read-side critical section, and even all at the same time, if need be. Doing so might not be wise from a contention or scalability viewpoint, but then again, doing so does not raise any fundamental conceptual issues.

So, what can TM do about time delays within transactions?

1. Ignore time delays within transactions. This has an appearance of elegance, but like too many other "elegant" solutions, fails to survive first contact with legacy code. Such code, which might well have important time delays in critical sections, would fail upon being transactionalized.

2. Abort transactions upon encountering a time-delay operation. This is attractive, but it is unfortunately not always possible to automatically detect a time-delay operation. Is that tight loop carrying out a critical computation, or is it simply waiting for time to elapse?

www.dbooks.org

3. Enlist the compiler to prohibit time delays within transactions.

4. Let the time delays execute normally. Unfortunately, some TM implementations publish modifications only at commit time, which could defeat the purpose of the time delay.

It is not clear that there is a single correct answer. TM implementations featuring weak atomicity that publish changes immediately within the transaction (rolling these changes back upon abort) might be reasonably well served by the last alternative. Even in this case, the code (or possibly even hardware) at the other end of the transaction may require a substantial redesign to tolerate aborted transactions. This need for redesign would make it more difficult to apply transactional memory to legacy code.

17.2.1.4 Persistence

There are many different types of locking primitives. One interesting distinction is persistence, in other words, whether the lock can exist independently of the address space of the process using the lock.

Non-persistent locks include `pthread_mutex_lock()`, `pthread_rwlock_rdlock()`, and most kernel-level locking primitives. If the memory locations instantiating a non-persistent lock's data structures disappear, so does the lock. For typical use of `pthread_mutex_lock()`, this means that when the process exits, all of its locks vanish. This property can be exploited in order to trivialize lock cleanup at program shutdown time, but makes it more difficult for unrelated applications to share locks, as such sharing requires the applications to share memory.

Quick Quiz 17.1: But suppose that an application exits while holding a `pthread_mutex_lock()` that happens to be located in a file-mapped region of memory? ■

Persistent locks help avoid the need to share memory among unrelated applications. Persistent locking APIs include the flock family, `lockf()`, System V semaphores, or the `O_CREAT` flag to `open()`. These persistent APIs can be used to protect large-scale operations spanning runs of multiple applications, and, in the case of `O_CREAT` even surviving operating-system reboot. If need be, locks can even span multiple computer systems via distributed lock managers and distributed filesystems—and persist across reboots of any or all of those computer systems.

Persistent locks can be used by any application, including applications written using multiple languages and software environments. In fact, a persistent lock might well be acquired by an application written in C and released by an application written in Python.

How could a similar persistent functionality be provided for TM?

1. Restrict persistent transactions to special-purpose environments designed to support them, for example, SQL. This clearly works, given the decades-long history of database systems, but does not provide the same degree of flexibility provided by persistent locks.

2. Use snapshot facilities provided by some storage devices and/or filesystems. Unfortunately, this does not handle network communication, nor does it handle I/O to devices that do not provide snapshot capabilities, for example, memory sticks.

3. Build a time machine.

4. Avoid the problem entirely by using existing persistent facilities, presumably avoiding such use within transactions.

Of course, the fact that it is called transactional *memory* should give us pause, as the name itself conflicts with the concept of a persistent transaction. It is nevertheless worthwhile to consider this possibility as an important test case probing the inherent limitations of transactional memory.

17.2.2 Process Modification

Processes are not eternal: They are created and destroyed, their memory mappings are modified, they are linked to dynamic libraries, and they are debugged. These sections look at how transactional memory can handle an ever-changing execution environment.

17.2.2.1 Multithreaded Transactions

It is perfectly legal to create processes and threads while holding a lock or, for that matter, while holding a hazard pointer, within a sequence-locking read-side critical section, and from within a userspace-RCU read-side critical section, and even all at the same time, if need be. Not only is it legal, but it is quite simple, as can be seen from the following code fragment:

```
1 pthread_mutex_lock(...);
2 for (i = 0; i < ncpus; i++)
```

```
3   pthread_create(&tid[i], ...);
4 for (i = 0; i < ncpus; i++)
5   pthread_join(tid[i], ...);
6 pthread_mutex_unlock(...);
```

This pseudo-code fragment uses `pthread_create()` to spawn one thread per CPU, then uses `pthread_join()` to wait for each to complete, all under the protection of `pthread_mutex_lock()`. The effect is to execute a lock-based critical section in parallel, and one could obtain a similar effect using `fork()` and `wait()`. Of course, the critical section would need to be quite large to justify the thread-spawning overhead, but there are many examples of large critical sections in production software.

What might TM do about thread spawning within a transaction?

1. Declare `pthread_create()` to be illegal within transactions, preferably by aborting the transaction. Alternatively, enlist the compiler to enforce `pthread_create()`-free transactions.

2. Permit `pthread_create()` to be executed within a transaction, but only the parent thread will be considered to be part of the transaction. This approach seems to be reasonably compatible with existing and posited TM implementations, but seems to be a trap for the unwary. This approach raises further questions, such as how to handle conflicting child-thread accesses.

3. Convert the `pthread_create()`s to function calls. This approach is also an attractive nuisance, as it does not handle the not-uncommon cases where the child threads communicate with one another. In addition, it does not permit concurrent execution of the body of the transaction.

4. Extend the transaction to cover the parent and all child threads. This approach raises interesting questions about the nature of conflicting accesses, given that the parent and children are presumably permitted to conflict with each other, but not with other threads. It also raises interesting questions as to what should happen if the parent thread does not wait for its children before committing the transaction. Even more interesting, what happens if the parent conditionally executes `pthread_join()` based on the values of variables participating in the transaction? The answers to these questions are reasonably straightforward in the case of locking. The answers for TM are left as an exercise for the reader.

Given that parallel execution of transactions is commonplace in the database world, it is perhaps surprising that current TM proposals do not provide for it. On the other hand, the example above is a fairly sophisticated use of locking that is not normally found in simple textbook examples, so perhaps its omission is to be expected. That said, some researchers are using transactions to autoparallelize code [RKM+10], and there are rumors that other TM researchers are investigating fork/join parallelism within transactions, so perhaps this topic will soon be addressed more thoroughly.

17.2.2.2 The `exec()` System Call

One can execute an `exec()` system call within a lock-based critical section, while holding a hazard pointer, within a sequence-locking read-side critical section, and from within a userspace-RCU read-side critical section, and even all at the same time, if need be. The exact semantics depends on the type of primitive.

In the case of non-persistent primitives (including `pthread_mutex_lock()`, `pthread_rwlock_rdlock()`, and userspace RCU), if the `exec()` succeeds, the whole address space vanishes, along with any locks being held. Of course, if the `exec()` fails, the address space still lives, so any associated locks would also still live. A bit strange perhaps, but well defined.

On the other hand, persistent primitives (including the flock family, `lockf()`, System V semaphores, and the `O_CREAT` flag to `open()`) would survive regardless of whether the `exec()` succeeded or failed, so that the `exec()`ed program might well release them.

Quick Quiz 17.2: What about non-persistent primitives represented by data structures in `mmap()` regions of memory? What happens when there is an `exec()` within a critical section of such a primitive? ■

What happens when you attempt to execute an `exec()` system call from within a transaction?

1. Disallow `exec()` within transactions, so that the enclosing transactions abort upon encountering the `exec()`. This is well defined, but clearly requires non-TM synchronization primitives for use in conjunction with `exec()`.

2. Disallow `exec()` within transactions, with the compiler enforcing this prohibition. There is a draft specification for TM in C++ that takes this approach, allowing functions to be decorated with the

www.dbooks.org

`transaction_safe` and `transaction_unsafe` attributes.[4] This approach has some advantages over aborting the transaction at runtime, but again requires non-TM synchronization primitives for use in conjunction with `exec()`. One disadvantage is the need to decorate a great many library functions with `transaction_safe` and `transaction_unsafe` attributes.

3. Treat the transaction in a manner similar to non-persistent locking primitives, so that the transaction survives if `exec()` fails, and silently commits if the `exec()` succeeds. The case where only some of the variables affected by the transaction reside in `mmap()`ed memory (and thus could survive a successful `exec()` system call) is left as an exercise for the reader.

4. Abort the transaction (and the `exec()` system call) if the `exec()` system call would have succeeded, but allow the transaction to continue if the `exec()` system call would fail. This is in some sense the "correct" approach, but it would require considerable work for a rather unsatisfying result.

The `exec()` system call is perhaps the strangest example of an obstacle to universal TM applicability, as it is not completely clear what approach makes sense, and some might argue that this is merely a reflection of the perils of real-life interaction with `exec()`. That said, the two options prohibiting `exec()` within transactions are perhaps the most logical of the group.

Similar issues surround the `exit()` and `kill()` system calls, as well as a `longjmp()` or an exception that would exit the transaction. (Where did the `longjmp()` or exception come from?)

17.2.2.3 Dynamic Linking and Loading

Lock-based critical section, code holding a hazard pointer, sequence-locking read-side critical sections, and userspace-RCU read-side critical sections can (separately or in combination) legitimately contain code that invokes dynamically linked and loaded functions, including C/C++ shared libraries and Java class libraries. Of course, the code contained in these libraries is by definition unknowable at compile time. So, what happens if a dynamically loaded function is invoked within a transaction?

[4] Thanks to Mark Moir for pointing me at this spec, and to Michael Wong for having pointed me at an earlier revision some time back.

This question has two parts: (a) how do you dynamically link and load a function within a transaction and (b) what do you do about the unknowable nature of the code within this function? To be fair, item (b) poses some challenges for locking and userspace-RCU as well, at least in theory. For example, the dynamically linked function might introduce a deadlock for locking or might (erroneously) introduce a quiescent state into a userspace-RCU read-side critical section. The difference is that while the class of operations permitted in locking and userspace-RCU critical sections is well-understood, there appears to still be considerable uncertainty in the case of TM. In fact, different implementations of TM seem to have different restrictions.

So what can TM do about dynamically linked and loaded library functions? Options for part (a), the actual loading of the code, include the following:

1. Treat the dynamic linking and loading in a manner similar to a page fault, so that the function is loaded and linked, possibly aborting the transaction in the process. If the transaction is aborted, the retry will find the function already present, and the transaction can thus be expected to proceed normally.

2. Disallow dynamic linking and loading of functions from within transactions.

Options for part (b), the inability to detect TM-unfriendly operations in a not-yet-loaded function, possibilities include the following:

1. Just execute the code: if there are any TM-unfriendly operations in the function, simply abort the transaction. Unfortunately, this approach makes it impossible for the compiler to determine whether a given group of transactions may be safely composed. One way to permit composability regardless is irrevocable transactions, however, current implementations permit only a single irrevocable transaction to proceed at any given time, which can severely limit performance and scalability. Irrevocable transactions also to restrict use of manual transaction-abort operations. Finally, if there is an irrevocable transaction manipulating a given data item, any other transaction manipulating that same data item cannot have non-blocking semantics.

2. Decorate the function declarations indicating which functions are TM-friendly. These decorations can then be enforced by the compiler's type system. Of

course, for many languages, this requires language extensions to be proposed, standardized, and implemented, with the corresponding time delays, and also with the corresponding decoration of a great many otherwise uninvolved library functions. That said, the standardization effort is already in progress [ATS09].

3. As above, disallow dynamic linking and loading of functions from within transactions.

I/O operations are of course a known weakness of TM, and dynamic linking and loading can be thought of as yet another special case of I/O. Nevertheless, the proponents of TM must either solve this problem, or resign themselves to a world where TM is but one tool of several in the parallel programmer's toolbox. (To be fair, a number of TM proponents have long since resigned themselves to a world containing more than just TM.)

17.2.2.4 Memory-Mapping Operations

It is perfectly legal to execute memory-mapping operations (including `mmap()`, `shmat()`, and `munmap()` [Gro01]) within a lock-based critical section, while holding a hazard pointer, within a sequence-locking read-side critical section, and from within a userspace-RCU read-side critical section, and even all at the same time, if need be. What happens when you attempt to execute such an operation from within a transaction? More to the point, what happens if the memory region being remapped contains some variables participating in the current thread's transaction? And what if this memory region contains variables participating in some other thread's transaction?

It should not be necessary to consider cases where the TM system's metadata is remapped, given that most locking primitives do not define the outcome of remapping their lock variables.

Here are some TM memory-mapping options:

1. Memory remapping is illegal within a transaction, and will result in all enclosing transactions being aborted. This does simplify things somewhat, but also requires that TM interoperate with synchronization primitives that do tolerate remapping from within their critical sections.

2. Memory remapping is illegal within a transaction, and the compiler is enlisted to enforce this prohibition.

3. Memory mapping is legal within a transaction, but aborts all other transactions having variables in the region mapped over.

4. Memory mapping is legal within a transaction, but the mapping operation will fail if the region being mapped overlaps with the current transaction's footprint.

5. All memory-mapping operations, whether within or outside a transaction, check the region being mapped against the memory footprint of all transactions in the system. If there is overlap, then the memory-mapping operation fails.

6. The effect of memory-mapping operations that overlap the memory footprint of any transaction in the system is determined by the TM conflict manager, which might dynamically determine whether to fail the memory-mapping operation or abort any conflicting transactions.

It is interesting to note that `munmap()` leaves the relevant region of memory unmapped, which could have additional interesting implications.[5]

17.2.2.5 Debugging

The usual debugging operations such as breakpoints work normally within lock-based critical sections and from usespace-RCU read-side critical sections. However, in initial transactional-memory hardware implementations [DLMN09] an exception within a transaction will abort that transaction, which in turn means that breakpoints abort all enclosing transactions.

So how can transactions be debugged?

1. Use software emulation techniques within transactions containing breakpoints. Of course, it might be necessary to emulate all transactions any time a breakpoint is set within the scope of any transaction. If the runtime system is unable to determine whether or not a given breakpoint is within the scope of a transaction, then it might be necessary to emulate all transactions just to be on the safe side. However, this approach might impose significant overhead, which might in turn obscure the bug being pursued.

[5] This difference between mapping and unmapping was noted by Josh Triplett.

www.dbooks.org

2. Use only hardware TM implementations that are capable of handling breakpoint exceptions. Unfortunately, as of this writing (March 2021), all such implementations are research prototypes.

3. Use only software TM implementations, which are (very roughly speaking) more tolerant of exceptions than are the simpler of the hardware TM implementations. Of course, software TM tends to have higher overhead than hardware TM, so this approach may not be acceptable in all situations.

4. Program more carefully, so as to avoid having bugs in the transactions in the first place. As soon as you figure out how to do this, please do let everyone know the secret!

There is some reason to believe that transactional memory will deliver productivity improvements compared to other synchronization mechanisms, but it does seem quite possible that these improvements could easily be lost if traditional debugging techniques cannot be applied to transactions. This seems especially true if transactional memory is to be used by novices on large transactions. In contrast, macho "top-gun" programmers might be able to dispense with such debugging aids, especially for small transactions.

Therefore, if transactional memory is to deliver on its productivity promises to novice programmers, the debugging problem does need to be solved.

17.2.3 Synchronization

If transactional memory someday proves that it can be everything to everyone, it will not need to interact with any other synchronization mechanism. Until then, it will need to work with synchronization mechanisms that can do what it cannot, or that work more naturally in a given situation. The following sections outline the current challenges in this area.

17.2.3.1 Locking

It is commonplace to acquire locks while holding other locks, which works quite well, at least as long as the usual well-known software-engineering techniques are employed to avoid deadlock. It is not unusual to acquire locks from within RCU read-side critical sections, which eases deadlock concerns because RCU read-side primitives cannot participate in lock-based deadlock cycles. It is also possible to acquire locks while holding hazard pointers and within sequence-lock read-side critical sections. But what happens when you attempt to acquire a lock from within a transaction?

In theory, the answer is trivial: simply manipulate the data structure representing the lock as part of the transaction, and everything works out perfectly. In practice, a number of non-obvious complications [VGS08] can arise, depending on implementation details of the TM system. These complications can be resolved, but at the cost of a 45 % increase in overhead for locks acquired outside of transactions and a 300 % increase in overhead for locks acquired within transactions. Although these overheads might be acceptable for transactional programs containing small amounts of locking, they are often completely unacceptable for production-quality lock-based programs wishing to use the occasional transaction.

1. Use only locking-friendly TM implementations. Unfortunately, the locking-unfriendly implementations have some attractive properties, including low overhead for successful transactions and the ability to accommodate extremely large transactions.

2. Use TM only "in the small" when introducing TM to lock-based programs, thereby accommodating the limitations of locking-friendly TM implementations.

3. Set aside locking-based legacy systems entirely, reimplementing everything in terms of transactions. This approach has no shortage of advocates, but this requires that all the issues described in this series be resolved. During the time it takes to resolve these issues, competing synchronization mechanisms will of course also have the opportunity to improve.

4. Use TM strictly as an optimization in lock-based systems, as was done by the TxLinux [RHP+07] group and by a great many transactional lock elision projects [PD11, Kle14, FIMR16, PMDY20]. This approach seems sound, but leaves the locking design constraints (such as the need to avoid deadlock) firmly in place.

5. Strive to reduce the overhead imposed on locking primitives.

The fact that there could possibly be a problem interfacing TM and locking came as a surprise to many, which underscores the need to try out new mechanisms and primitives in real-world production software. Fortunately, the advent of open source means that a huge quantity of such software is now freely available to everyone, including researchers.

17.2.3.2 Reader-Writer Locking

It is commonplace to read-acquire reader-writer locks while holding other locks, which just works, at least as long as the usual well-known software-engineering techniques are employed to avoid deadlock. Read-acquiring reader-writer locks from within RCU read-side critical sections also works, and doing so eases deadlock concerns because RCU read-side primitives cannot participate in lock-based deadlock cycles. It is also possible to acquire locks while holding hazard pointers and within sequence-lock read-side critical sections. But what happens when you attempt to read-acquire a reader-writer lock from within a transaction?

Unfortunately, the straightforward approach to read-acquiring the traditional counter-based reader-writer lock within a transaction defeats the purpose of the reader-writer lock. To see this, consider a pair of transactions concurrently attempting to read-acquire the same reader-writer lock. Because read-acquisition involves modifying the reader-writer lock's data structures, a conflict will result, which will roll back one of the two transactions. This behavior is completely inconsistent with the reader-writer lock's goal of allowing concurrent readers.

Here are some options available to TM:

1. Use per-CPU or per-thread reader-writer locking [HW92], which allows a given CPU (or thread, respectively) to manipulate only local data when read-acquiring the lock. This would avoid the conflict between the two transactions concurrently read-acquiring the lock, permitting both to proceed, as intended. Unfortunately, (1) the write-acquisition overhead of per-CPU/thread locking can be extremely high, (2) the memory overhead of per-CPU/thread locking can be prohibitive, and (3) this transformation is available only when you have access to the source code in question. Other more-recent scalable reader-writer locks [LLO09] might avoid some or all of these problems.

2. Use TM only "in the small" when introducing TM to lock-based programs, thereby avoiding read-acquiring reader-writer locks from within transactions.

3. Set aside locking-based legacy systems entirely, re-implementing everything in terms of transactions. This approach has no shortage of advocates, but this requires that *all* the issues described in this series be resolved. During the time it takes to resolve these issues, competing synchronization mechanisms will of course also have the opportunity to improve.

4. Use TM strictly as an optimization in lock-based systems, as was done by the TxLinux [RHP+07] group, and as has been done by more recent work using TM to elide reader writer locks [FIMR16]. This approach seems sound, at least on POWER8 CPUs [LGW+15], but leaves the locking design constraints (such as the need to avoid deadlock) firmly in place.

Of course, there might well be other non-obvious issues surrounding combining TM with reader-writer locking, as there in fact were with exclusive locking.

17.2.3.3 Deferred Reclamation

This section focuses mainly on RCU. Similar issues and possible resolutions arise when combining TM with other deferred-reclamation mechanisms such as reference counters and hazard pointers. In the text below, known differences are specifically called out.

Reference counting, hazard pointers, and RCU are all heavily used, as noted in Sections 9.5.5 and 9.6.3. This means that any TM implementation that chooses not to surmount each and every challenge called out in this section needs to interoperate cleanly and efficiently with all of these synchronization mechanisms.

The TxLinux group from the University of Texas at Austin appears to be the group to take on the challenge of RCU/TM interoperability [RHP+07]. Because they applied TM to the Linux 2.6 kernel, which uses RCU, they had no choice but to integrate TM and RCU, with TM taking the place of locking for RCU updates. Unfortunately, although the paper does state that the RCU implementation's locks (e.g., `rcu_ctrlblk.lock`) were converted to transactions, it is silent about what was done with those locks used by RCU-based updates (for example, `dcache_lock`).

More recently, Dimitrios Siakavaras et al. have applied HTM and RCU to search trees [SNGK17, SBN+20], Christina Giannoula et al. have used HTM and RCU to color graphs [GGK18], and SeongJae Park et al. have used HTM and RCU to optimize high-contention locking on NUMA systems [PMDY20].

It is important to note that RCU permits readers and updaters to run concurrently, further permitting RCU readers to access data that is in the act of being updated. Of course, this property of RCU, whatever its performance, scalability, and real-time-response benefits might be, flies

www.dbooks.org

in the face of the underlying atomicity properties of TM, although the POWER8 CPU family's suspended-transaction facility [LGW+15] makes it an exception to this rule.

So how should TM-based updates interact with concurrent RCU readers? Some possibilities are as follows:

1. RCU readers abort concurrent conflicting TM updates. This is in fact the approach taken by the TxLinux project. This approach does preserve RCU semantics, and also preserves RCU's read-side performance, scalability, and real-time-response properties, but it does have the unfortunate side-effect of unnecessarily aborting conflicting updates. In the worst case, a long sequence of RCU readers could potentially starve all updaters, which could in theory result in system hangs. In addition, not all TM implementations offer the strong atomicity required to implement this approach, and for good reasons.

2. RCU readers that run concurrently with conflicting TM updates get old (pre-transaction) values from any conflicting RCU loads. This preserves RCU semantics and performance, and also prevents RCU-update starvation. However, not all TM implementations can provide timely access to old values of variables that have been tentatively updated by an in-flight transaction. In particular, log-based TM implementations that maintain old values in the log (thus providing excellent TM commit performance) are not likely to be happy with this approach. Perhaps the `rcu_dereference()` primitive can be leveraged to permit RCU to access the old values within a greater range of TM implementations, though performance might still be an issue. Nevertheless, there are popular TM implementations that have been integrated with RCU in this manner [PW07, HW11, HW13].

3. If an RCU reader executes an access that conflicts with an in-flight transaction, then that RCU access is delayed until the conflicting transaction either commits or aborts. This approach preserves RCU semantics, but not RCU's performance or real-time response, particularly in presence of long-running transactions. In addition, not all TM implementations are capable of delaying conflicting accesses. Nevertheless, this approach seems eminently reasonable for hardware TM implementations that support only small transactions.

4. RCU readers are converted to transactions. This approach pretty much guarantees that RCU is compatible with any TM implementation, but it also imposes TM's rollbacks on RCU read-side critical sections, destroying RCU's real-time response guarantees, and also degrading RCU's read-side performance. Furthermore, this approach is infeasible in cases where any of the RCU read-side critical sections contains operations that the TM implementation in question is incapable of handling. This approach is more difficult to apply to hazard pointers and reference counters, which do not have a sharply defined notion of a reader as a section of code.

5. Many update-side uses of RCU modify a single pointer to publish a new data structure. In some of these cases, RCU can safely be permitted to see a transactional pointer update that is subsequently rolled back, as long as the transaction respects memory ordering and as long as the roll-back process uses `call_rcu()` to free up the corresponding structure. Unfortunately, not all TM implementations respect memory barriers within a transaction. Apparently, the thought is that because transactions are supposed to be atomic, the ordering of the accesses within the transaction is not supposed to matter.

6. Prohibit use of TM in RCU updates. This is guaranteed to work, but restricts use of TM.

It seems likely that additional approaches will be uncovered, especially given the advent of user-level RCU and hazard-pointer implementations.[6] It is interesting to note that many of the better performing and scaling STM implementations make use of RCU-like techniques internally [Fra04, FH07, GYW+19, KMK+19].

Quick Quiz 17.3: MV-RLU looks pretty good! Doesn't it beat RCU hands down? ■

17.2.3.4 Extra-Transactional Accesses

Within a lock-based critical section, it is perfectly legal to manipulate variables that are concurrently accessed or even modified outside that lock's critical section, with one common example being statistical counters. The same thing is possible within RCU read-side critical sections, and is in fact the common case.

[6] Kudos to the TxLinux group, Maged Michael, and Josh Triplett for coming up with a number of the above alternatives.

Given mechanisms such as the so-called "dirty reads" that are prevalent in production database systems, it is not surprising that extra-transactional accesses have received serious attention from the proponents of TM, with the concept of weak atomicity [BLM06] being but one case in point.

Here are some extra-transactional options:

1. Conflicts due to extra-transactional accesses always abort transactions. This is strong atomicity.
2. Conflicts due to extra-transactional accesses are ignored, so only conflicts among transactions can abort transactions. This is weak atomicity.
3. Transactions are permitted to carry out non-transactional operations in special cases, such as when allocating memory or interacting with lock-based critical sections.
4. Produce hardware extensions that permit some operations (for example, addition) to be carried out concurrently on a single variable by multiple transactions.
5. Introduce weak semantics to transactional memory. One approach is the combination with RCU described in Section 17.2.3.3, while Gramoli and Guerraoui survey a number of other weak-transaction approaches [GG14], for example, restricted partitioning of large "elastic" transactions into smaller transactions, thus reducing conflict probabilities (albeit with tepid performance and scalability). Perhaps further experience will show that some uses of extra-transactional accesses can be replaced by weak transactions.

It appears that transactions were conceived in a vacuum, with no interaction required with any other synchronization mechanism. If so, it is no surprise that much confusion and complexity arises when combining transactions with non-transactional accesses. But unless transactions are to be confined to small updates to isolated data structures, or alternatively to be confined to new programs that do not interact with the huge body of existing parallel code, then transactions absolutely must be so combined if they are to have large-scale practical impact in the near term.

17.2.4 Discussion

The obstacles to universal TM adoption lead to the following conclusions:

1. One interesting property of TM is the fact that transactions are subject to rollback and retry. This property underlies TM's difficulties with irreversible operations, including unbuffered I/O, RPCs, memory-mapping operations, time delays, and the `exec()` system call. This property also has the unfortunate consequence of introducing all the complexities inherent in the possibility of failure, often in a developer-visible manner.
2. Another interesting property of TM, noted by Shpeisman et al. [SATG+09], is that TM intertwines the synchronization with the data it protects. This property underlies TM's issues with I/O, memory-mapping operations, extra-transactional accesses, and debugging breakpoints. In contrast, conventional synchronization primitives, including locking and RCU, maintain a clear separation between the synchronization primitives and the data that they protect.
3. One of the stated goals of many workers in the TM area is to ease parallelization of large sequential programs. As such, individual transactions are commonly expected to execute serially, which might do much to explain TM's issues with multithreaded transactions.

Quick Quiz 17.4: Given things like `spin_trylock()`, how does it make any sense at all to claim that TM introduces the concept of failure??? ■

What should TM researchers and developers do about all of this?

One approach is to focus on TM in the small, focusing on small transactions where hardware assist potentially provides substantial advantages over other synchronization primitives and on small programs where there is some evidence for increased productivity for a combined TM-locking approach [PAT11]. Sun took the small-transaction approach with its Rock research CPU [DLMN09]. Some TM researchers seem to agree with these two small-is-beautiful approaches [SSHT93], others have much higher hopes for TM, and yet others hint that high TM aspirations might be TM's worst enemy [Att10, Section 6]. It is nonetheless quite possible that TM will be able to take on larger problems, and this section has listed a few of the issues that must be resolved if TM is to achieve this lofty goal.

Of course, everyone involved should treat this as a learning experience. It would seem that TM researchers

www.dbooks.org

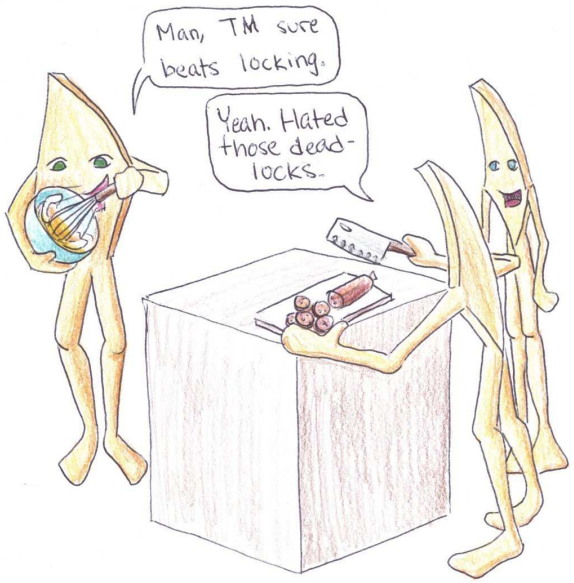


Figure 17.9: The STM Vision



Figure 17.10: The STM Reality: Conflicts

have great deal to learn from practitioners who have successfully built large software systems using traditional synchronization primitives.

And vice versa.

Quick Quiz 17.5: What is to learn? Why not just use TM for memory-based data structures and locking for those rare cases featuring the many silly corner cases listed in this silly section??? ■

But for the moment, the current state of STM can best be summarized with a series of cartoons. First, Figure 17.9 shows the STM vision. As always, the reality is a bit more nuanced, as fancifully depicted by Figures 17.10, 17.11, and 17.12.[7] Less fanciful STM retrospectives are also available [Duf10a, Duf10b].

Some commercially available hardware supports restricted variants of HTM, which are addressed in the following section.



Figure 17.11: The STM Reality: Irrevocable Operations

[7] Recent academic work-in-progress has investigated lock-based STM systems for real-time use [And19, NA18], albeit without any performance results, and with some indications that real-time hybrid STM/HTM systems must choose between fast common-case performance and worst-case forward-progress guarantees [AKK+14, SBV10].



Figure 17.12: The STM Reality: Realtime Response

17.3 Hardware Transactional Memory

> Make sure your report system is reasonably clean and efficient before you automate. Otherwise, your new computer will just speed up the mess.
>
> *Robert Townsend*

As of 2021, hardware transactional memory (HTM) has been available for many years on several types of commercially available commodity computer systems [YHLR13, Mer11, JSG12, Hay20]. This section makes an attempt to identify HTM's place in the parallel programmer's toolbox.

From a conceptual viewpoint, HTM uses processor caches and speculative execution to make a designated group of statements (a "transaction") take effect atomically from the viewpoint of any other transactions running on other processors. This transaction is initiated by a begin-transaction machine instruction and completed by a commit-transaction machine instruction. There is typically also an abort-transaction machine instruction, which squashes the speculation (as if the begin-transaction instruction and all following instructions had not executed) and commences execution at a failure handler. The location of the failure handler is typically specified by the begin-transaction instruction, either as an explicit failure-handler address or via a condition code set by the instruction itself. Each transaction executes atomically with respect to all other transactions.

HTM has a number of important benefits, including automatic dynamic partitioning of data structures, reducing synchronization-primitive cache misses, and supporting a fair number of practical applications.

However, it always pays to read the fine print, and HTM is no exception. A major point of this section is determining under what conditions HTM's benefits outweigh the complications hidden in its fine print. To this end, Section 17.3.1 describes HTM's benefits and Section 17.3.2 describes its weaknesses. This is the same approach used in earlier papers [MMW07, MMTW10] and also in the previous section.[8]

Section 17.3.3 then describes HTM's weaknesses with respect to the combination of synchronization primitives used in the Linux kernel (and in many user-space applications). Section 17.3.4 looks at where HTM might best fit into the parallel programmer's toolbox, and Section 17.3.5 lists some events that might greatly increase HTM's scope and appeal. Finally, Section 17.3.6 presents concluding remarks.

17.3.1 HTM Benefits WRT to Locking

The primary benefits of HTM are (1) its avoidance of the cache misses that are often incurred by other synchronization primitives, (2) its ability to dynamically partition data structures, and (3) the fact that it has a fair number of practical applications. I break from TM tradition by not listing ease of use separately for two reasons. First, ease of use should stem from HTM's primary benefits, which this section focuses on. Second, there has been considerable controversy surrounding attempts to test for raw programming talent [Bor06, DBA09, PBCE20] and even around the use of small programming exercises in job interviews [Bra07]. This indicates that we really do not have a firm grasp on what makes programming easy or hard. Therefore, the remainder of this section focuses on the three benefits listed above.

17.3.1.1 Avoiding Synchronization Cache Misses

Most synchronization mechanisms are based on data structures that are operated on by atomic instructions. Because these atomic instructions normally operate by first causing the relevant cache line to be owned by the CPU that they are

[8] I gratefully acknowledge many stimulating discussions with the other authors, Maged Michael, Josh Triplett, and Jonathan Walpole, as well as with Andi Kleen.

www.dbooks.org

running on, a subsequent execution of the same instance of that synchronization primitive on some other CPU will result in a cache miss. These communications cache misses severely degrade both the performance and scalability of conventional synchronization mechanisms [ABD+97, Section 4.2.3].

In contrast, HTM synchronizes by using the CPU's cache, avoiding the need for a separate synchronization data structure and resultant cache misses. HTM's advantage is greatest in cases where a lock data structure is placed in a separate cache line, in which case, converting a given critical section to an HTM transaction can reduce that critical section's overhead by a full cache miss. These savings can be quite significant for the common case of short critical sections, at least for those situations where the elided lock does not share a cache line with an oft-written variable protected by that lock.

Quick Quiz 17.6: Why would it matter that oft-written variables shared the cache line with the lock variable? ■

17.3.1.2 Dynamic Partitioning of Data Structures

A major obstacle to the use of some conventional synchronization mechanisms is the need to statically partition data structures. There are a number of data structures that are trivially partitionable, with the most prominent example being hash tables, where each hash chain constitutes a partition. Allocating a lock for each hash chain then trivially parallelizes the hash table for operations confined to a given chain.[9] Partitioning is similarly trivial for arrays, radix trees, skiplists, and several other data structures.

However, partitioning for many types of trees and graphs is quite difficult, and the results are often quite complex [Ell80]. Although it is possible to use two-phased locking and hashed arrays of locks to partition general data structures, other techniques have proven preferable [Mil06], as will be discussed in Section 17.3.3. Given its avoidance of synchronization cache misses, HTM is therefore a very real possibility for large non-partitionable data structures, at least assuming relatively small updates.

Quick Quiz 17.7: Why are relatively small updates important to HTM performance and scalability? ■

[9] And it is also easy to extend this scheme to operations accessing multiple hash chains by having such operations acquire the locks for all relevant chains in hash order.

17.3.1.3 Practical Value

Some evidence of HTM's practical value has been demonstrated in a number of hardware platforms, including Sun Rock [DLMN09], Azul Vega [Cli09], IBM Blue Gene/Q [Mer11], and Intel Haswell TSX [RD12], IBM System z [JSG12].

Expected practical benefits include:

1. Lock elision for in-memory data access and update [MT01, RG02].
2. Concurrent access and small random updates to large non-partitionable data structures.

However, HTM also has some very real shortcomings, which will be discussed in the next section.

17.3.2 HTM Weaknesses WRT Locking

The concept of HTM is quite simple: A group of accesses and updates to memory occurs atomically. However, as is the case with many simple ideas, complications arise when you apply it to real systems in the real world. These complications are as follows:

1. Transaction-size limitations.
2. Conflict handling.
3. Aborts and rollbacks.
4. Lack of forward-progress guarantees.
5. Irrevocable operations.
6. Semantic differences.

Each of these complications is covered in the following sections, followed by a summary.

17.3.2.1 Transaction-Size Limitations

The transaction-size limitations of current HTM implementations stem from the use of the processor caches to hold the data affected by the transaction. Although this allows a given CPU to make the transaction appear atomic to other CPUs by executing the transaction within the confines of its cache, it also means that any transaction that does not fit cannot commit. Furthermore, events that change execution context, such as interrupts, system calls, exceptions, traps, and context switches either must abort any ongoing transaction on the CPU in question or must

further restrict transaction size due to the cache footprint of the other execution context.

Of course, modern CPUs tend to have large caches, and the data required for many transactions would fit easily in a one-megabyte cache. Unfortunately, with caches, sheer size is not all that matters. The problem is that most caches can be thought of hash tables implemented in hardware. However, hardware caches do not chain their buckets (which are normally called *sets*), but rather provide a fixed number of cachelines per set. The number of elements provided for each set in a given cache is termed that cache's *associativity*.

Although cache associativity varies, the eight-way associativity of the level-0 cache on the laptop I am typing this on is not unusual. What this means is that if a given transaction needed to touch nine cache lines, and if all nine cache lines mapped to the same set, then that transaction cannot possibly complete, never mind how many megabytes of additional space might be available in that cache. Yes, given randomly selected data elements in a given data structure, the probability of that transaction being able to commit is quite high, but there can be no guarantee [McK11c].

There has been some research work to alleviate this limitation. Fully associative *victim caches* would alleviate the associativity constraints, but there are currently stringent performance and energy-efficiency constraints on the sizes of victim caches. That said, HTM victim caches for unmodified cache lines can be quite small, as they need to retain only the address: The data itself can be written to memory or shadowed by other caches, while the address itself is sufficient to detect a conflicting write [RD12].

Unbounded transactional memory (UTM) schemes [AAKL06, MBM+06] use DRAM as an extremely large victim cache, but integrating such schemes into a production-quality cache-coherence mechanism is still an unsolved problem. In addition, use of DRAM as a victim cache may have unfortunate performance and energy-efficiency consequences, particularly if the victim cache is to be fully associative. Finally, the "unbounded" aspect of UTM assumes that all of DRAM could be used as a victim cache, while in reality the large but still fixed amount of DRAM assigned to a given CPU would limit the size of that CPU's transactions. Other schemes use a combination of hardware and software transactional memory [KCH+06] and one could imagine using STM as a fallback mechanism for HTM.

However, to the best of my knowledge, with the exception of abbreviating representation of TM read sets, currently available systems do not implement any of these research ideas, and perhaps for good reason.

17.3.2.2 Conflict Handling

The first complication is the possibility of *conflicts*. For example, suppose that transactions A and B are defined as follows:

```
Transaction A       Transaction B

x = 1;              y = 2;
y = 3;              x = 4;
```

Suppose that each transaction executes concurrently on its own processor. If transaction A stores to `x` at the same time that transaction B stores to `y`, neither transaction can progress. To see this, suppose that transaction A executes its store to `y`. Then transaction A will be interleaved within transaction B, in violation of the requirement that transactions execute atomically with respect to each other. Allowing transaction B to execute its store to `x` similarly violates the atomic-execution requirement. This situation is termed a *conflict*, which happens whenever two concurrent transactions access the same variable where at least one of the accesses is a store. The system is therefore obligated to abort one or both of the transactions in order to allow execution to progress. The choice of exactly which transaction to abort is an interesting topic that will very likely retain the ability to generate Ph.D. dissertations for some time to come, see for example [ATC+11].[10] For the purposes of this section, we can assume that the system makes a random choice.

Another complication is conflict detection, which is comparatively straightforward, at least in the simplest case. When a processor is executing a transaction, it marks every cache line touched by that transaction. If the processor's cache receives a request involving a cache line that has been marked as touched by the current transaction, a potential conflict has occurred. More sophisticated systems might try to order the current processors' transaction to precede that of the processor sending the request, and optimizing this process will likely also retain the ability to generate Ph.D. dissertations for quite some time. However this section assumes a very simple conflict-detection strategy.

However, for HTM to work effectively, the probability of conflict must be quite low, which in turn requires that the data structures be organized so as to maintain a sufficiently low probability of conflict. For example, a

[10] Liu's and Spear's paper entitled "Toxic Transactions" [LS11] is particularly instructive.

www.dbooks.org

red-black tree with simple insertion, deletion, and search operations fits this description, but a red-black tree that maintains an accurate count of the number of elements in the tree does not.[11] For another example, a red-black tree that enumerates all elements in the tree in a single transaction will have high conflict probabilities, degrading performance and scalability. As a result, many serial programs will require some restructuring before HTM can work effectively. In some cases, practitioners will prefer to take the extra steps (in the red-black-tree case, perhaps switching to a partitionable data structure such as a radix tree or a hash table), and just use locking, particularly until such time as HTM is readily available on all relevant architectures [Cli09].

Quick Quiz 17.8: How could a red-black tree possibly efficiently enumerate all elements of the tree regardless of choice of synchronization mechanism??? ■

Furthermore, the potential for conflicting accesses among concurrent transactions can result in failure failure. Handling such failure is discussed in the next section.

17.3.2.3 Aborts and Rollbacks

Because any transaction might be aborted at any time, it is important that transactions contain no statements that cannot be rolled back. This means that transactions cannot do I/O, system calls, or debugging breakpoints (no single stepping in the debugger for HTM transactions!!!). Instead, transactions must confine themselves to accessing normal cached memory. Furthermore, on some systems, interrupts, exceptions, traps, TLB misses, and other events will also abort transactions. Given the number of bugs that have resulted from improper handling of error conditions, it is fair to ask what impact aborts and rollbacks have on ease of use.

Quick Quiz 17.9: But why can't a debugger emulate single stepping by setting breakpoints at successive lines of the transaction, relying on the retry to retrace the steps of the earlier instances of the transaction? ■

Of course, aborts and rollbacks raise the question of whether HTM can be useful for hard real-time systems. Do the performance benefits of HTM outweigh the costs of the aborts and rollbacks, and if so under what conditions? Can transactions use priority boosting? Or should transactions for high-priority threads instead preferentially abort those of low-priority threads? If so, how is the hardware efficiently informed of priorities? The literature on real-time use of HTM is quite sparse, perhaps because there are more than enough problems in making HTM work well in non-real-time environments.

Because current HTM implementations might deterministically abort a given transaction, software must provide fallback code. This fallback code must use some other form of synchronization, for example, locking. If a lock-based fallback is ever used, then all the limitations of locking, including the possibility of deadlock, reappear. One can of course hope that the fallback isn't used often, which might allow simpler and less deadlock-prone locking designs to be used. But this raises the question of how the system transitions from using the lock-based fallbacks back to transactions.[12] One approach is to use a test-and-test-and-set discipline [MT02], so that everyone holds off until the lock is released, allowing the system to start from a clean slate in transactional mode at that point. However, this could result in quite a bit of spinning, which might not be wise if the lock holder has blocked or been preempted. Another approach is to allow transactions to proceed in parallel with a thread holding a lock [MT02], but this raises difficulties in maintaining atomicity, especially if the reason that the thread is holding the lock is because the corresponding transaction would not fit into cache.

Finally, dealing with the possibility of aborts and rollbacks seems to put an additional burden on the developer, who must correctly handle all combinations of possible error conditions.

It is clear that users of HTM must put considerable validation effort into testing both the fallback code paths and transition from fallback code back to transactional code. Nor is there any reason to believe that the validation requirements of HTM hardware are any less daunting.

17.3.2.4 Lack of Forward-Progress Guarantees

Even though transaction size, conflicts, and aborts/rollbacks can all cause transactions to abort, one might hope that sufficiently small and short-duration transactions could be guaranteed to eventually succeed. This would permit a transaction to be unconditionally retried, in the same way that compare-and-swap (CAS) and load-linked/store-conditional (LL/SC) operations are unconditionally retried

[11] The need to update the count would result in additions to and deletions from the tree conflicting with each other, resulting in strong non-commutativity [AGH+11a, AGH+11b, McK11b].

[12] The possibility of an application getting stuck in fallback mode has been termed the "lemming effect", a term that Dave Dice has been credited with coining.

in code that uses these instructions to implement atomic operations.

Unfortunately, other than low-clock-rate academic research prototypes [SBV10], currently available HTM implementations refuse to make any sort of forward-progress guarantee. As noted earlier, HTM therefore cannot be used to avoid deadlock on those systems. Hopefully future implementations of HTM will provide some sort of forward-progress guarantees. Until that time, HTM must be used with extreme caution in real-time applications.

The one exception to this gloomy picture as of 2021 is the IBM mainframe, which provides *constrained transactions* [JSG12]. The constraints are quite severe, and are presented in Section 17.3.5.1. It will be interesting to see if HTM forward-progress guarantees migrate from the mainframe to commodity CPU families.

17.3.2.5 Irrevocable Operations

Another consequence of aborts and rollbacks is that HTM transactions cannot accommodate irrevocable operations. Current HTM implementations typically enforce this limitation by requiring that all of the accesses in the transaction be to cacheable memory (thus prohibiting MMIO accesses) and aborting transactions on interrupts, traps, and exceptions (thus prohibiting system calls).

Note that buffered I/O can be accommodated by HTM transactions as long as the buffer fill/flush operations occur extra-transactionally. The reason that this works is that adding data to and removing data from the buffer is revocable: Only the actual buffer fill/flush operations are irrevocable. Of course, this buffered-I/O approach has the effect of including the I/O in the transaction's footprint, increasing the size of the transaction and thus increasing the probability of failure.

17.3.2.6 Semantic Differences

Although HTM can in many cases be used as a drop-in replacement for locking (hence the name transactional lock elision [DHL+08]), there are subtle differences in semantics. A particularly nasty example involving coordinated lock-based critical sections that results in deadlock or livelock when executed transactionally was given by Blundell [BLM06], but a much simpler example is the empty critical section.

In a lock-based program, an empty critical section will guarantee that all processes that had previously been holding that lock have now released it. This idiom was used by the 2.4 Linux kernel's networking stack to coordinate changes in configuration. But if this empty critical section is translated to a transaction, the result is a no-op. The guarantee that all prior critical sections have terminated is lost. In other words, transactional lock elision preserves the data-protection semantics of locking, but loses locking's time-based messaging semantics.

Quick Quiz 17.10: But why would *anyone* need an empty lock-based critical section??? ■

Quick Quiz 17.11: Can't transactional lock elision trivially handle locking's time-based messaging semantics by simply choosing not to elide empty lock-based critical sections? ■

Quick Quiz 17.12: Given modern hardware [MOZ09], how can anyone possibly expect parallel software relying on timing to work? ■

One important semantic difference between locking and transactions is the priority boosting that is used to avoid priority inversion in lock-based real-time programs. One way in which priority inversion can occur is when a low-priority thread holding a lock is preempted by a medium-priority CPU-bound thread. If there is at least one such medium-priority thread per CPU, the low-priority thread will never get a chance to run. If a high-priority thread now attempts to acquire the lock, it will block. It cannot acquire the lock until the low-priority thread releases it, the low-priority thread cannot release the lock until it gets a chance to run, and it cannot get a chance to run until one of the medium-priority threads gives up its CPU. Therefore, the medium-priority threads are in effect blocking the high-priority process, which is the rationale for the name "priority inversion."

One way to avoid priority inversion is *priority inheritance*, in which a high-priority thread blocked on a lock temporarily donates its priority to the lock's holder, which is also called *priority boosting*. However, priority boosting can be used for things other than avoiding priority inversion, as shown in Listing 17.1. Lines 1–12 of this listing show a low-priority process that must nevertheless run every millisecond or so, while lines 14–24 of this same listing show a high-priority process that uses priority boosting to ensure that `boostee()` runs periodically as needed.

The `boostee()` function arranges this by always holding one of the two `boost_lock[]` locks, so that lines 20–21 of `booster()` can boost priority as needed.

Quick Quiz 17.13: But the `boostee()` function in Listing 17.1 alternatively acquires its locks in reverse order! Won't this result in deadlock? ■

www.dbooks.org

Listing 17.1: Exploiting Priority Boosting

```
1  void boostee(void)
2  {
3    int i = 0;
4
5    acquire_lock(&boost_lock[i]);
6    for (;;) {
7      acquire_lock(&boost_lock[!i]);
8      release_lock(&boost_lock[i]);
9      i = i ^ 1;
10     do_something();
11   }
12 }
13
14 void booster(void)
15 {
16   int i = 0;
17
18   for (;;) {
19     usleep(500); /* sleep 0.5 ms. */
20     acquire_lock(&boost_lock[i]);
21     release_lock(&boost_lock[i]);
22     i = i ^ 1;
23   }
24 }
```

This arrangement requires that `boostee()` acquire its first lock on line 5 before the system becomes busy, but this is easily arranged, even on modern hardware.

Unfortunately, this arrangement can break down in presence of transactional lock elision. The `boostee()` function's overlapping critical sections become one infinite transaction, which will sooner or later abort, for example, on the first time that the thread running the `boostee()` function is preempted. At this point, `boostee()` will fall back to locking, but given its low priority and that the quiet initialization period is now complete (which after all is why `boostee()` was preempted), this thread might never again get a chance to run.

And if the `boostee()` thread is not holding the lock, then the `booster()` thread's empty critical section on lines 20 and 21 of Listing 17.1 will become an empty transaction that has no effect, so that `boostee()` never runs. This example illustrates some of the subtle consequences of transactional memory's rollback-and-retry semantics.

Given that experience will likely uncover additional subtle semantic differences, application of HTM-based lock elision to large programs should be undertaken with caution. That said, where it does apply, HTM-based lock elision can eliminate the cache misses associated with the lock variable, which has resulted in tens of percent performance increases in large real-world software systems as of early 2015. We can therefore expect to see substantial use of this technique on hardware supporting it.

Quick Quiz 17.14: So a bunch of people set out to supplant locking, and they mostly end up just optimizing locking??? ■

17.3.2.7 Summary

Although it seems likely that HTM will have compelling use cases, current implementations have serious transaction-size limitations, conflict-handling complications, abort-and-rollback issues, and semantic differences that will require careful handling. HTM's current situation relative to locking is summarized in Table 17.1. As can be seen, although the current state of HTM alleviates some serious shortcomings of locking,[13] it does so by introducing a significant number of shortcomings of its own. These shortcomings are acknowledged by leaders in the TM community [MS12].[14]

In addition, this is not the whole story. Locking is not normally used by itself, but is instead typically augmented by other synchronization mechanisms, including reference counting, atomic operations, non-blocking data structures, hazard pointers [Mic04, HLM02], and RCU [MS98a, MAK+01, HMBW07, McK12b]. The next section looks at how such augmentation changes the equation.

17.3.3 HTM Weaknesses WRT to Locking When Augmented

Practitioners have long used reference counting, atomic operations, non-blocking data structures, hazard pointers, and RCU to avoid some of the shortcomings of locking. For example, deadlock can be avoided in many cases by using reference counts, hazard pointers, or RCU to protect data structures, particularly for read-only critical sections [Mic04, HLM02, DMS+12, GMTW08, HMBW07]. These approaches also reduce the need to partition data structures, as was seen in Chapter 10. RCU further provides contention-free bounded wait-free read-side primitives [MS98a, DMS+12], while hazard pointers provides

[13] In fairness, it is important to emphasize that locking's shortcomings do have well-known and heavily used engineering solutions, including deadlock detectors [Cor06a], a wealth of data structures that have been adapted to locking, and a long history of augmentation, as discussed in Section 17.3.3. In addition, if locking really were as horrible as a quick skim of many academic papers might reasonably lead one to believe, where did all the large lock-based parallel programs (both FOSS and proprietary) come from, anyway?

[14] In addition, in early 2011, I was invited to deliver a critique of some of the assumptions underlying transactional memory [McK11e]. The audience was surprisingly non-hostile, though perhaps they were taking it easy on me due to the fact that I was heavily jet-lagged while giving the presentation.

Table 17.1: Comparison of Locking and HTM (Advantage , Disadvantage , Strong Disadvantage)

	Locking	Hardware Transactional Memory
Basic Idea	Allow only one thread at a time to access a given set of objects.	Cause a given operation over a set of objects to execute atomically.
Scope	Handles all operations.	Handles revocable operations.
		Irrevocable operations force fallback (typically to locking).
Composability	Limited by deadlock.	Limited by irrevocable operations, transaction size, and deadlock (assuming lock-based fallback code).
Scalability & Performance	Data must be partitionable to avoid lock contention.	Data must be partitionable to avoid conflicts.
	Partioning must typically be fixed at design time.	Dynamic adjustment of partitioning carried out automatically down to cacheline boundaries.
		Partitioning required for fallbacks (less important for rare fallbacks).
	Locking primitives typically result in expensive cache misses and memory-barrier instructions.	Transactions begin/end instructions typically do not result in cache misses, but do have memory-ordering and overhead consequences.
	Contention effects are focused on acquisition and release, so that the critical section runs at full speed.	Contention aborts conflicting transactions, even if they have been running for a long time.
	Privatization operations are simple, intuitive, performant, and scalable.	Privatized data contributes to transaction size.
Hardware Support	Commodity hardware suffices.	New hardware required (and is starting to become available).
	Performance is insensitive to cache-geometry details.	Performance depends critically on cache geometry.
Software Support	APIs exist, large body of code and experience, debuggers operate naturally.	APIs emerging, little experience outside of DBMS, breakpoints mid-transaction can be problematic.
Interaction With Other Mechanisms	Long experience of successful interaction.	Just beginning investigation of interaction.
Practical Apps	Yes.	Yes.
Wide Applicability	Yes.	Jury still out.

www.dbooks.org

Table 17.2: Comparison of Locking (Augmented by RCU or Hazard Pointers) and HTM (Advantage , Disadvantage , Strong Disadvantage)

	Locking with Userspace RCU or Hazard Pointers	Hardware Transactional Memory
Basic Idea	Allow only one thread at a time to access a given set of objects.	Cause a given operation over a set of objects to execute atomically.
Scope	Handles all operations.	Handles revocable operations.
		Irrevocable operations force fallback (typically to locking).
Composability	Readers limited only by grace-period-wait operations.	Limited by irrevocable operations, transaction size, and deadlock. (Assuming lock-based fallback code.)
	Updaters limited by deadlock. Readers reduce deadlock.	
Scalability & Performance	Data must be partitionable to avoid lock contention among updaters.	Data must be partitionable to avoid conflicts.
	Partitioning not needed for readers.	
	Partitioning for updaters must typically be fixed at design time.	Dynamic adjustment of partitioning carried out automatically down to cacheline boundaries.
	Partitioning not needed for readers.	Partitioning required for fallbacks (less important for rare fallbacks).
	Updater locking primitives typically result in expensive cache misses and memory-barrier instructions.	Transactions begin/end instructions typically do not result in cache misses, but do have memory-ordering and overhead consequences.
	Update-side contention effects are focused on acquisition and release, so that the critical section runs at full speed.	Contention aborts conflicting transactions, even if they have been running for a long time.
	Readers do not contend with updaters or with each other.	
	Read-side primitives are typically bounded wait-free with low overhead. (Lock-free with low overhead for hazard pointers.)	Read-only transactions subject to conflicts and rollbacks. No forward-progress guarantees other than those supplied by fallback code.
	Privatization operations are simple, intuitive, performant, and scalable when data is visible only to updaters.	Privatized data contributes to transaction size.
	Privatization operations are expensive (though still intuitive and scalable) for reader-visible data.	
Hardware Support	Commodity hardware suffices.	New hardware required (and is starting to become available).
	Performance is insensitive to cache-geometry details.	Performance depends critically on cache geometry.
Software Support	APIs exist, large body of code and experience, debuggers operate naturally.	APIs emerging, little experience outside of DBMS, breakpoints mid-transaction can be problematic.
Interaction With Other Mechanisms	Long experience of successful interaction.	Just beginning investigation of interaction.
Practical Apps	Yes.	Yes.
Wide Applicability	Yes.	Jury still out.

lock-free read-side primitives [Mic02, HLM02, Mic04]. Adding these considerations to Table 17.1 results in the updated comparison between augmented locking and HTM shown in Table 17.2. A summary of the differences between the two tables is as follows:

1. Use of non-blocking read-side mechanisms alleviates deadlock issues.
2. Read-side mechanisms such as hazard pointers and RCU can operate efficiently on non-partitionable data.
3. Hazard pointers and RCU do not contend with each other or with updaters, allowing excellent performance and scalability for read-mostly workloads.
4. Hazard pointers and RCU provide forward-progress guarantees (lock freedom and bounded wait-freedom, respectively).
5. Privatization operations for hazard pointers and RCU are straightforward.

For those with good eyesight, Table 17.3 combines Tables 17.1 and 17.2.

Of course, it is also possible to augment HTM, as discussed in the next section.

17.3.4 Where Does HTM Best Fit In?

Although it will likely be some time before HTM's area of applicability can be as crisply delineated as that shown for RCU in Figure 9.30 on page 166, that is no reason not to start moving in that direction.

HTM seems best suited to update-heavy workloads involving relatively small changes to disparate portions of relatively large in-memory data structures running on large multiprocessors, as this meets the size restrictions of current HTM implementations while minimizing the probability of conflicts and attendant aborts and rollbacks. This scenario is also one that is relatively difficult to handle given current synchronization primitives.

Use of locking in conjunction with HTM seems likely to overcome HTM's difficulties with irrevocable operations, while use of RCU or hazard pointers might alleviate HTM's transaction-size limitations for read-only operations that traverse large fractions of the data structure [PMDY20]. Current HTM implementations unconditionally abort an update transaction that conflicts with an RCU or hazard-pointer reader, but perhaps future HTM implementations will interoperate more smoothly with these synchronization mechanisms. In the meantime, the probability of an update conflicting with a large RCU or hazard-pointer read-side critical section should be much smaller than the probability of conflicting with the equivalent read-only transaction.[15] Nevertheless, it is quite possible that a steady stream of RCU or hazard-pointer readers might starve updaters due to a corresponding steady stream of conflicts. This vulnerability could be eliminated (at significant hardware cost and complexity) by giving extra-transactional reads the pre-transaction copy of the memory location being loaded.

The fact that HTM transactions must have fallbacks might in some cases force static partitionability of data structures back onto HTM. This limitation might be alleviated if future HTM implementations provide forward-progress guarantees, which might eliminate the need for fallback code in some cases, which in turn might allow HTM to be used efficiently in situations with higher conflict probabilities.

In short, although HTM is likely to have important uses and applications, it is another tool in the parallel programmer's toolbox, not a replacement for the toolbox in its entirety.

17.3.5 Potential Game Changers

Game changers that could greatly increase the need for HTM include the following:

1. Forward-progress guarantees.
2. Transaction-size increases.
3. Improved debugging support.
4. Weak atomicity.

These are expanded upon in the following sections.

17.3.5.1 Forward-Progress Guarantees

As was discussed in Section 17.3.2.4, current HTM implementations lack forward-progress guarantees, which requires that fallback software is available to handle HTM failures. Of course, it is easy to demand guarantees, but not always easy to provide them. In the case of HTM,

[15] It is quite ironic that strictly transactional mechanisms are appearing in shared-memory systems at just about the time that NoSQL databases are relaxing the traditional database-application reliance on strict transactions. Nevertheless, HTM has in fact realized the ease-of-use promise of TM, albeit for black-hat attacks on the Linux kernel's address-space randomization defense mechanism [JLK16a, JLK16b].

www.dbooks.org

Table 17.3: Comparison of Locking (Plain and Augmented) and HTM (Advantage , Disadvantage , Strong Disadvantage)

	Locking	Locking with Userspace RCU or Hazard Pointers	Hardware Transactional Memory
Basic Idea	Allow only one thread at a time to access a given set of objects.	Allow only one thread at a time to access a given set of objects.	Cause a given operation over a set of objects to execute atomically.
Scope	Handles all operations.	Handles all operations.	Handles revocable operations.
			Irrevocable operations force fallback (typically to locking).
Composability	Limited by deadlock.	Readers limited only by grace-period-wait operations.	Limited by irrevocable operations, transaction size, and deadlock. (Assuming lock-based fallback code.)
		Updaters limited by deadlock. Readers reduce deadlock.	
Scalability & Performance	Data must be partitionable to avoid lock contention.	Data must be partitionable to avoid lock contention among updaters.	Data must be partitionable to avoid conflicts.
		Partitioning not needed for readers.	
	Partitioning must typically be fixed at design time.	Partitioning for updaters must typically be fixed at design time.	Dynamic adjustment of partitioning carried out automatically down to cacheline boundaries.
		Partitioning not needed for readers.	Partitioning required for fallbacks (less important for rare fallbacks).
	Locking primitives typically result in expensive cache misses and memory-barrier instructions.	Updater locking primitives typically result in expensive cache misses and memory-barrier instructions.	Transactions begin/end instructions typically do not result in cache misses, but do have memory-ordering and overhead consequences.
	Contention effects are focused on acquisition and release, so that the critical section runs at full speed.	Update-side contention effects are focused on acquisition and release, so that the critical section runs at full speed.	Contention aborts conflicting transactions, even if they have been running for a long time.
		Readers do not contend with updaters or with each other.	
		Read-side primitives are typically bounded wait-free with low overhead. (Lock-free with low overhead for hazard pointers.)	Read-only transactions subject to conflicts and rollbacks. No forward-progress guarantees other than those supplied by fallback code.
	Privatization operations are simple, intuitive, performant, and scalable.	Privatization operations are simple, intuitive, performant, and scalable when data is visible only to updaters.	Privatized data contributes to transaction size.
		Privatization operations are expensive (though still intuitive and scalable) for reader-visible data.	
Hardware Support	Commodity hardware suffices.	Commodity hardware suffices.	New hardware required (and is starting to become available).
	Performance is insensitive to cache-geometry details.	Performance is insensitive to cache-geometry details.	Performance depends critically on cache geometry.
Software Support	APIs exist, large body of code and experience, debuggers operate naturally.	APIs exist, large body of code and experience, debuggers operate naturally.	APIs emerging, little experience outside of DBMS, breakpoints mid-transaction can be problematic.
Interaction With Other Mechanisms	Long experience of successful interaction.	Long experience of successful interaction.	Just beginning investigation of interaction.
Practical Apps	Yes.	Yes.	Yes.
Wide Applicability	Yes.	Yes.	Jury still out.

Edition.2-rc11

obstacles to guarantees can include cache size and associativity, TLB size and associativity, transaction duration and interrupt frequency, and scheduler implementation.

Cache size and associativity was discussed in Section 17.3.2.1, along with some research intended to work around current limitations. However, HTM forward-progress guarantees would come with size limits, large though these limits might one day be. So why don't current HTM implementations provide forward-progress guarantees for small transactions, for example, limited to the associativity of the cache? One potential reason might be the need to deal with hardware failure. For example, a failing cache SRAM cell might be handled by deactivating the failing cell, thus reducing the associativity of the cache and therefore also the maximum size of transactions that can be guaranteed forward progress. Given that this would simply decrease the guaranteed transaction size, it seems likely that other reasons are at work. Perhaps providing forward progress guarantees on production-quality hardware is more difficult than one might think, an entirely plausible explanation given the difficulty of making forward-progress guarantees in software. Moving a problem from software to hardware does not necessarily make it easier to solve [JSG12].

Given a physically tagged and indexed cache, it is not enough for the transaction to fit in the cache. Its address translations must also fit in the TLB. Any forward-progress guarantees must therefore also take TLB size and associativity into account.

Given that interrupts, traps, and exceptions abort transactions in current HTM implementations, it is necessary that the execution duration of a given transaction be shorter than the expected interval between interrupts. No matter how little data a given transaction touches, if it runs too long, it will be aborted. Therefore, any forward-progress guarantees must be conditioned not only on transaction size, but also on transaction duration.

Forward-progress guarantees depend critically on the ability to determine which of several conflicting transactions should be aborted. It is all too easy to imagine an endless series of transactions, each aborting an earlier transaction only to itself be aborted by a later transactions, so that none of the transactions actually commit. The complexity of conflict handling is evidenced by the large number of HTM conflict-resolution policies that have been proposed [ATC+11, LS11]. Additional complications are introduced by extra-transactional accesses, as noted by Blundell [BLM06]. It is easy to blame the extra-transactional accesses for all of these problems, but the folly of this line of thinking is easily demonstrated by placing each of the extra-transactional accesses into its own single-access transaction. It is the pattern of accesses that is the issue, not whether or not they happen to be enclosed in a transaction.

Finally, any forward-progress guarantees for transactions also depend on the scheduler, which must let the thread executing the transaction run long enough to successfully commit.

So there are significant obstacles to HTM vendors offering forward-progress guarantees. However, the impact of any of them doing so would be enormous. It would mean that HTM transactions would no longer need software fallbacks, which would mean that HTM could finally deliver on the TM promise of deadlock elimination.

However, in late 2012, the IBM Mainframe announced an HTM implementation that includes *constrained transactions* in addition to the usual best-effort HTM implementation [JSG12]. A constrained transaction starts with the `tbeginc` instruction instead of the `tbegin` instruction that is used for best-effort transactions. Constrained transactions are guaranteed to always complete (eventually), so if a transaction aborts, rather than branching to a fallback path (as is done for best-effort transactions), the hardware instead restarts the transaction at the `tbeginc` instruction.

The Mainframe architects needed to take extreme measures to deliver on this forward-progress guarantee. If a given constrained transaction repeatedly fails, the CPU might disable branch prediction, force in-order execution, and even disable pipelining. If the repeated failures are due to high contention, the CPU might disable speculative fetches, introduce random delays, and even serialize execution of the conflicting CPUs. "Interesting" forward-progress scenarios involve as few as two CPUs or as many as one hundred CPUs. Perhaps these extreme measures provide some insight as to why other CPUs have thus far refrained from offering constrained transactions.

As the name implies, constrained transactions are in fact severely constrained:

1. The maximum data footprint is four blocks of memory, where each block can be no larger than 32 bytes.
2. The maximum code footprint is 256 bytes.
3. If a given 4K page contains a constrained transaction's code, then that page may not contain that transaction's data.
4. The maximum number of assembly instructions that may be executed is 32.

www.dbooks.org

5. Backwards branches are forbidden.

Nevertheless, these constraints support a number of important data structures, including linked lists, stacks, queues, and arrays. Constrained HTM therefore seems likely to become an important tool in the parallel programmer's toolbox.

Note that these forward-progress guarantees need not be absolute. For example, suppose that a use of HTM uses a global lock as fallback. Assuming that the fallback mechanism has been carefully designed to avoid the "lemming effect" discussed in Section 17.3.2.3, then if HTM rollbacks are sufficiently infrequent, the global lock will not be a bottleneck. That said, the larger the system, the longer the critical sections, and the longer the time required to recover from the "lemming effect", the more rare "sufficiently infrequent" needs to be.

17.3.5.2 Transaction-Size Increases

Forward-progress guarantees are important, but as we saw, they will be conditional guarantees based on transaction size and duration. There has been some progress, for example, some commercially available HTM implementations use approximation techniques to support extremely large HTM read sets [RD12]. For another example, POWER8 HTM supports suspended transations, which avoid adding irrelevant accesses to the suspended transation's read and write sets [LGW+15]. This capability has been used to produce a high performance reader-writer lock [FIMR16].

It is important to note that even small-sized guarantees will be quite useful. For example, a guarantee of two cache lines is sufficient for a stack, queue, or dequeue. However, larger data structures require larger guarantees, for example, traversing a tree in order requires a guarantee equal to the number of nodes in the tree. Therefore, even modest increases in the size of the guarantee also increases the usefulness of HTM, thereby increasing the need for CPUs to either provide it or provide good-and-sufficient workarounds.

17.3.5.3 Improved Debugging Support

Another inhibitor to transaction size is the need to debug the transactions. The problem with current mechanisms is that a single-step exception aborts the enclosing transaction. There are a number of workarounds for this issue, including emulating the processor (slow!), substituting STM for HTM (slow and slightly different semantics!), playback techniques using repeated retries to emulate forward progress (strange failure modes!), and full support of debugging HTM transactions (complex!).

Should one of the HTM vendors produce an HTM system that allows straightforward use of classical debugging techniques within transactions, including breakpoints, single stepping, and print statements, this will make HTM much more compelling. Some transactional-memory researchers started to recognize this problem in 2013, with at least one proposal involving hardware-assisted debugging facilities [GKP13]. Of course, this proposal depends on readily available hardware gaining such facilities [Hay20, Int20]. Worse yet, some cutting-edge debugging facilities are incompatible with HTM [OHOC20].

17.3.5.4 Weak Atomicity

Given that HTM is likely to face some sort of size limitations for the foreseeable future, it will be necessary for HTM to interoperate smoothly with other mechanisms. HTM's interoperability with read-mostly mechanisms such as hazard pointers and RCU would be improved if extra-transactional reads did not unconditionally abort transactions with conflicting writes—instead, the read could simply be provided with the pre-transaction value. In this way, hazard pointers and RCU could be used to allow HTM to handle larger data structures and to reduce conflict probabilities.

This is not necessarily simple, however. The most straightforward way of implementing this requires an additional state in each cache line and on the bus, which is a non-trivial added expense. The benefit that goes along with this expense is permitting large-footprint readers without the risk of starving updaters due to continual conflicts. An alternative approach, applied to great effect to binary search trees by Siakavaras et al. [SNGK17], is to use RCU for read-only traversals and HTM only for the actual updates themselves. This combination outperformed other transactional-memory techniques by up to 220 %, a speedup similar to that observed by Howard and Walpole [HW11] when they combined RCU with STM. In both cases, the weak atomicity is implemented in software rather than in hardware. It would nevertheless be interesting to see what additional speedups could be obtained by implementing weak atomicity in both hardware and software.

17.3.6 Conclusions

Although current HTM implementations have delivered real performance benefits in some situations, they also have significant shortcomings. The most significant shortcomings appear to be limited transaction sizes, the need for conflict handling, the need for aborts and rollbacks, the lack of forward-progress guarantees, the inability to handle irrevocable operations, and subtle semantic differences from locking.

Some of these shortcomings might be alleviated in future implementations, but it appears that there will continue to be a strong need to make HTM work well with the many other types of synchronization mechanisms, as noted earlier [MMW07, MMTW10]. Although there has been some work using HTM with RCU [SNGK17, SBN+20, GGK18, PMDY20], there has been little evidence of work making HTM work better with RCU and with other deferred-reclamation mechanisms.

In short, current HTM implementations appear to be welcome and useful additions to the parallel programmer's toolbox, and much interesting and challenging work is required to make use of them. However, they cannot be considered to be a magic wand with which to wave away all parallel-programming problems.

17.4 Formal Regression Testing?

> Theory without experiments: Have we gone too far?
>
> *Michael Mitzenmacher*

Formal verification has long proven useful in a number of production environments [LBD+04, BBC+10, Coo18, SAE+18, DFLO19]. However, it is an question as to whether hard-core formal verification will ever be included in the automated regression-test suites used for continuous integration within complex concurrent codebases, such as the Linux kernel. Although there is already a proof of concept for Linux-kernel SRCU [Roy17], this test is for a small portion of one of the simplest RCU implementations, and has proven difficult to keep it current with the ever-changing Linux kernel. It is therefore worth asking what would be required to incorporate formal verification as first-class members of the Linux kernel's regression tests.

The following list is a good start [McK15a, slide 34]:

1. Any required translation must be automated.
2. The environment (including memory ordering) must be correctly handled.
3. The memory and CPU overhead must be acceptably modest.
4. Specific information leading to the location of the bug must be provided.
5. Information beyond the source code and inputs must be modest in scope.
6. The bugs located must be relevant to the code's users.

This list builds on, but is somewhat more modest than, Richard Bornat's dictum: "Formal-verification researchers should verify the code that developers write, in the language they write it in, running in the environment that it runs in, as they write it." The following sections discuss each of the above requirements, followed by a section presenting a scorecard of how well a few tools stack up against these requirements.

17.4.1 Automatic Translation

Although Promela and `spin` are invaluable design aids, if you need to formally regression-test your C-language program, you must hand-translate to Promela each time you would like to re-verify your code. If your code happens to be in the Linux kernel, which releases every 60–90 days, you will need to hand-translate from four to six times each year. Over time, human error will creep in, which means that the verification won't match the source code, rendering the verification useless. Repeated verification clearly requires either that the formal-verification tooling input your code directly, or that there be bug-free automatic translation of your code to the form required for verification.

PPCMEM and `herd` can in theory directly input assembly language and C++ code, but these tools work only on very small litmus tests, which normally means that you must extract the core of your mechanism—by hand. As with Promela and `spin`, both PPCMEM and `herd` are extremely useful, but they are not well-suited for regression suites.

In contrast, `cbmc` and Nidhugg can input C programs of reasonable (though still quite limited) size, and if their capabilities continue to grow, could well become excellent additions to regression suites. The Coverity static-analysis tool also inputs C programs, and of very large size, including the Linux kernel. Of course, Coverity's static analysis is quite simple compared to that of

www.dbooks.org

cbmc and Nidhugg. On the other hand, Coverity had an all-encompassing definition of "C program" that posed special challenges [BBC+10]. Amazon Web Services uses a variety of formal-verification tool, including cbmc, and applies some of these tools to regression testing [Coo18]. Google uses a number of relatively simple static analysis tools directly on large Java code bases, which are arguably less diverse than C code bases [SAE+18]. Facebook uses more aggressive forms of formal verification against its code bases, including analysis of concurrency [DFLO19, O'H19], though not yet on the Linux kernel. Finally, Microsoft has long used static analysis on its code bases [LBD+04].

Given this list, it is clearly possible to create sophisticated formal-verification tools that directly consume production-quality source code.

However, one shortcoming of taking C code as input is that it assumes that the compiler is correct. An alternative approach is to take the binary produced by the C compiler as input, thereby accounting for any relevant compiler bugs. This approach has been used in a number of verification efforts, perhaps most notably by the SEL4 project [SM13].

Quick Quiz 17.15: Given the groundbreaking nature of the various verifiers used in the SEL4 project, why doesn't this chapter cover them in more depth? ■

However, verifying directly from either the source or binary both have the advantage of eliminating human translation errors, which is critically important for reliable regression testing.

This is not to say that tools with special-purpose languages are useless. On the contrary, they can be quite helpful for design-time verification, as was discussed in Chapter 12. However, such tools are not particularly helpful for automated regression testing, which is in fact the topic of this section.

17.4.2 Environment

It is critically important that formal-verification tools correctly model their environment. One all-too-common omission is the memory model, where a great many formal-verification tools, including Promela/spin, are restricted to sequential consistency. The QRCU experience related in Section 12.1.4.6 is an important cautionary tale.

Promela and spin assume sequential consistency, which is not a good match for modern computer systems, as was seen in Chapter 15. In contrast, one of the great strengths of PPCMEM and herd is their detailed modeling of various CPU families memory models, including x86, Arm, Power, and, in the case of herd, a Linux-kernel memory model [AMM+18], which was accepted into Linux-kernel version v4.17.

The cbmc and Nidhugg tools provide some ability to select memory models, but do not provide the variety that PPCMEM and herd do. However, it is likely that the larger-scale tools will adopt a greater variety of memory models as time goes on.

In the longer term, it would be helpful for formal-verification tools to include I/O [MDR16], but it may be some time before this comes to pass.

Nevertheless, tools that fail to match the environment can still be useful. For example, a great many concurrency bugs would still be bugs on a mythical sequentially consistent system, and these bugs could be located by a tool that over-approximates the system's memory model with sequential consistency. Nevertheless, these tools will fail to find bugs involving missing memory-ordering directives, as noted in the aforementioned cautionary tale of Section 12.1.4.6.

17.4.3 Overhead

Almost all hard-core formal-verification tools are exponential in nature, which might seem discouraging until you consider that many of the most interesting software questions are in fact undecidable. However, there are differences in degree, even among exponentials.

PPCMEM by design is unoptimized, in order to provide greater assurance that the memory models of interest are accurately represented. The herd tool optimizes more aggressively, as described in Section 12.3, and is thus orders of magnitude faster than PPCMEM. Nevertheless, both PPCMEM and herd target very small litmus tests rather than larger bodies of code.

In contrast, Promela/spin, cbmc, and Nidhugg are designed for (somewhat) larger bodies of code. Promela/spin was used to verify the Curiosity rover's filesystem [GHH+14] and, as noted earlier, both cbmc and Nidhugg were appled to Linux-kernel RCU.

If advances in heuristics continue at the rate of the past three decades, we can look forward to large reductions in overhead for formal verification. That said, combinatorial explosion is still combinatorial explosion, which would be expected to sharply limit the size of programs that could be verified, with or without continued improvements in heuristics.

However, the flip side of combinatorial explosion is Philip II of Macedon's timeless advice: "Divide and rule." If a large program can be divided and the pieces verified,

Listing 17.2: Emulating Locking with `cmpxchg_acquire()`

```
1  C C-SB+l-o-o-u+l-o-o-u-C
2
3  {}
4
5  P0(int *sl, int *x0, int *x1)
6  {
7    int r2;
8    int r1;
9
10   r2 = cmpxchg_acquire(sl, 0, 1);
11   WRITE_ONCE(*x0, 1);
12   r1 = READ_ONCE(*x1);
13   smp_store_release(sl, 0);
14 }
15
16 P1(int *sl, int *x0, int *x1)
17 {
18   int r2;
19   int r1;
20
21   r2 = cmpxchg_acquire(sl, 0, 1);
22   WRITE_ONCE(*x1, 1);
23   r1 = READ_ONCE(*x0);
24   smp_store_release(sl, 0);
25 }
26
27 filter (0:r2=0 /\ 1:r2=0)
28 exists (0:r1=0 /\ 1:r1=0)
```

the result can be combinatorial *implosion* [McK11e]. One natural place to divide is on API boundaries, for example, those of locking primitives. One verification pass can then verify that the locking implementation is correct, and additional verification passes can verify correct use of the locking APIs.

Table 17.4: Emulating Locking: Performance (s)

# Threads	Locking	`cmpxchg_acquire`
2	0.004	0.022
3	0.041	0.743
4	0.374	59.565
5	4.905	

The performance benefits of this approach can be demonstrated using the Linux-kernel memory model [AMM+18]. This model provides `spin_lock()` and `spin_unlock()` primitives, but these primitives can also be emulated using `cmpxchg_acquire()` and `smp_store_release()`, as shown in Listing 17.2 (`C-SB+l-o-o-u+l-o-o-*u.litmus` and `C-SB+l-o-o-u+l-o-o-u*-C.litmus`). Table 17.4 compares the performance and scalability of using the model's `spin_lock()` and `spin_unlock()` against emulating these primitives as shown in the listing. The difference is not insignificant: At four processes, the model is more than two orders of magnitude faster than emulation!

Quick Quiz 17.16: Why bother with a separate `filter` command on line 27 of Listing 17.2 instead of just adding the condition to the `exists` clause? And wouldn't it be simpler to use `xchg_acquire()` instead of `cmpxchg_acquire()`? ■

It would of course be quite useful for tools to automatically divide up large programs, verify the pieces, and then verify the combinations of pieces. In the meantime, verification of large programs will require significant manual intervention. This intervention will preferably mediated by scripting, the better to reliably carry out repeated verifications on each release, and preferably eventually in a manner well-suited for continuous integration. And Facebook's Infer tool has taken important steps towards doing just that, via compositionality and abstraction [BGOS18, DFLO19].

In any case, we can expect formal-verification capabilities to continue to increase over time, and any such increases will in turn increase the applicability of formal verification to regression testing.

17.4.4 Locate Bugs

Any software artifact of any size contains bugs. Therefore, a formal-verification tool that reports only the presence or absence of bugs is not particularly useful. What is needed is a tool that gives at least *some* information as to where the bug is located and the nature of that bug.

The `cbmc` output includes a traceback mapping back to the source code, similar to Promela/spin's, as does Nidhugg. Of course, these tracebacks can be quite long, and analyzing them can be quite tedious. However, doing so is usually quite a bit faster and more pleasant than locating bugs the old-fashioned way.

In addition, one of the simplest tests of formal-verification tools is bug injection. After all, not only could any of us write `printf("VERIFIED\n")`, but the plain fact is that developers of formal-verification tools are just as bug-prone as are the rest of us. Therefore, formal-verification tools that just proclaim that a bug exists are fundamentally less trustworthy because it is more difficult to verify them on real-world code.

All that aside, people writing formal-verification tools are permitted to leverage existing tools. For example, a tool designed to determine only the presence or absence of a serious but rare bug might leverage bisection. If an old version of the program under test did not contain the bug, but a new version did, then bisection could be used to quickly locate the commit that inserted the bug, which might be sufficient information to find and fix the bug.

www.dbooks.org

Of course, this sort of strategy would not work well for common bugs because in this case bisection would fail due to all commits having at least one instance of the common bug.

Therefore, the execution traces provided by many formal-verification tools will continue to be valuable, particularly for complex and difficult-to-understand bugs. In addition, recent work applies *incorrectness-logic* formalism reminiscent of the traditional Hoare logic used for full-up correctness proofs, but with the sole purpose of finding bugs [O'H19].

17.4.5 Minimal Scaffolding

In the old days, formal-verification researchers demanded a full specification against which the software would be verified. Unfortunately, a mathematically rigorous specification might well be larger than the actual code, and each line of specification is just as likely to contain bugs as is each line of code. A formal verification effort proving that the code faithfully implemented the specification would be a proof of bug-for-bug compatibility between the two, which might not be all that helpful.

Worse yet, the requirements for a number of software artifacts, including Linux-kernel RCU, are empirical in nature [McK15g, McK15d, McK15e].[16] For this common type of software, a complete specification is a polite fiction. Nor are complete specifications any less fictional for hardware, as was made clear by the late-2017 Meltdown and Spectre side-channel attacks [Hor18].

This situation might cause one to give up all hope of formal verification of real-world software and hardware artifacts, but it turns out that there is quite a bit that can be done. For example, design and coding rules can act as a partial specification, as can assertions contained in the code. And in fact formal-verification tools such as `cbmc` and Nidhugg both check for assertions that can be triggered, implicitly treating these assertions as part of the specification. However, the assertions are also part of the code, which makes it less likely that they will become obsolete, especially if the code is also subjected to stress tests.[17] The `cbmc` tool also checks for array-out-of-bound references, thus implicitly adding them to the specification. The aforementioned incorrectness logic can also be thought of as using an implicit bugs-not-present specification [O'H19].

[16] Or, in formal-verification parlance, Linux-kernel RCU has an *incomplete specification*.

[17] And you *do* stress-test your code, don't you?

This implicit-specification approach makes quite a bit of sense, particularly if you look at formal verification not as a full proof of correctness, but rather an alternative form of validation with a different set of strengths and weaknesses than the common case, that is, testing. From this viewpoint, software will always have bugs, and therefore any tool of any kind that helps to find those bugs is a very good thing indeed.

17.4.6 Relevant Bugs

Finding bugs—and fixing them—is of course the whole point of any type of validation effort. Clearly, false positives are to be avoided. But even in the absence of false positives, there are bugs and there are bugs.

For example, suppose that a software artifact had exactly 100 remaining bugs, each of which manifested on average once every million years of runtime. Suppose further that an omniscient formal-verification tool located all 100 bugs, which the developers duly fixed. What happens to the reliability of this software artifact?

The answer is that the reliability *decreases*.

To see this, keep in mind that historical experience indicates that about 7 % of fixes introduce a new bug [BJ12]. Therefore, fixing the 100 bugs, which had a combined mean time to failure (MTBF) of about 10,000 years, will introduce seven more bugs. Historical statistics indicate that each new bug will have an MTBF much less than 70,000 years. This in turn suggests that the combined MTBF of these seven new bugs will most likely be much less than 10,000 years, which in turn means that the well-intentioned fixing of the original 100 bugs actually decreased the reliability of the overall software.

Quick Quiz 17.17: How do we know that the MTBFs of known bugs is a good estimate of the MTBFs of bugs that have not yet been located? ■

Quick Quiz 17.18: But the formal-verification tools should immediately find all the bugs introduced by the fixes, so why is this a problem? ■

Worse yet, imagine another software artifact with one bug that fails once every day on average and 99 more that fail every million years each. Suppose that a formal-verification tool located the 99 million-year bugs, but failed to find the one-day bug. Fixing the 99 bugs located will take time and effort, decrease reliability, and do nothing at all about the pressing each-day failure that is likely causing embarrassment and perhaps much worse besides.

Therefore, it would be best to have a validation tool that preferentially located the most troublesome bugs. However, as noted in Section 17.4.4, it is permissible to leverage additional tools. One powerful tool is none other than plain old testing. Given knowledge of the bug, it should be possible to construct specific tests for it, possibly also using some of the techniques described in Section 11.6.4 to increase the probability of the bug manifesting. These techniques should allow calculation of a rough estimate of the bug's raw failure rate, which could in turn be used to prioritize bug-fix efforts.

Quick Quiz 17.19: But many formal-verification tools can only find one bug at a time, so that each bug must be fixed before the tool can locate the next. How can bug-fix efforts be prioritized given such a tool? ■

There has been some recent formal-verification work that prioritizes executions having fewer preemptions, under that reasonable assumption that smaller numbers of preemptions are more likely.

Identifying relevant bugs might sound like too much to ask, but it is what is really required if we are to actually increase software reliability.

17.4.7 Formal Regression Scorecard

Table 17.5 shows a rough-and-ready scorecard for the formal-verification tools covered in this chapter. Shorter wavelengths are better than longer wavelengths.

Promela requires hand translation and supports only sequential consistency, so its first two cells are red. It has reasonable overhead (for formal verification, anyway) and provides a traceback, so its next two cells are yellow. Despite requiring hand translation, Promela handles assertions in a natural way, so its fifth cell is green.

PPCMEM usually requires hand translation due to the small size of litmus tests that it supports, so its first cell is orange. It handles several memory models, so its second cell is green. Its overhead is quite high, so its third cell is red. It provides a graphical display of relations among operations, which is not as helpful as a traceback, but is still quite useful, so its fourth cell is yellow. It requires constructing an `exists` clause and cannot take intra-process assertions, so its fifth cell is also yellow.

The `herd` tool has size restrictions similar to those of PPCMEM, so `herd`'s first cell is also orange. It supports a wide variety of memory models, so its second cell is blue. It has reasonable overhead, so its third cell is yellow. Its bug-location and assertion capabilities are quite similar to those of PPCMEM, so `herd` also gets yellow for the next two cells.

The `cbmc` tool inputs C code directly, so its first cell is blue. It supports a few memory models, so its second cell is yellow. It has reasonable overhead, so its third cell is also yellow, however, perhaps SAT-solver performance will continue improving. It provides a traceback, so its fourth cell is green. It takes assertions directly from the C code, so its fifth cell is blue.

Nidhugg also inputs C code directly, so its first cell is also blue. It supports only a couple of memory models, so its second cell is orange. Its overhead is quite low (for formal-verification), so its third cell is green. It provides a traceback, so its fourth cell is green. It takes assertions directly from the C code, so its fifth cell is blue.

So what about the sixth and final row? It is too early to tell how any of the tools do at finding the right bugs, so they are all yellow with question marks.

Quick Quiz 17.20: How would testing stack up in the scorecard shown in Table 17.5? ■

Quick Quiz 17.21: But aren't there a great many more formal-verification systems than are shown in Table 17.5? ■

Once again, please note that this table rates these tools for use in regression testing. Just because many of them are a poor fit for regression testing does not at all mean that they are useless, in fact, many of them have proven their worth many times over.[18] Just not for regression testing.

However, this might well change. After all, formal verification tools made impressive strides in the 2010s. If that progress continues, formal verification might well become an indispensible tool in the parallel programmer's validation toolbox.

17.5 Functional Programming for Parallelism

> The curious failure of functional programming for parallel applications.
>
> *Malte Skarupke*

When I took my first-ever functional-programming class in the early 1980s, the professor asserted that the side-

[18] For but one example, Promela was used to verify the file system of none other than the Curiosity Rover. Was *your* formal verification tool used software currently running on Mars???

www.dbooks.org

Table 17.5: Formal Regression Scorecard

	Promela	PPCMEM	herd	cbmc	Nidhugg
(1) Automated					
(2) Environment	(MM)			(MM)	(MM)
(3) Overhead				(SAT)	
(4) Locate Bugs					
(5) Minimal Scaffolding					
(6) Relevant Bugs	???	???	???	???	???

effect-free functional-programming style was well-suited to trivial parallelization and analysis. Thirty years later, this assertion remains, but mainstream production use of parallel functional languages is minimal, a state of affairs that might not be entirely unrelated to professor's additional assertion that programs should neither maintain state nor do I/O. There is niche use of functional languages such as Erlang, and multithreaded support has been added to several other functional languages, but mainstream production usage remains the province of procedural languages such as C, C++, Java, and Fortran (usually augmented with OpenMP, MPI, or coarrays).

This situation naturally leads to the question "If analysis is the goal, why not transform the procedural language into a functional language before doing the analysis?" There are of course a number of objections to this approach, of which I list but three:

1. Procedural languages often make heavy use of global variables, which can be updated independently by different functions, or, worse yet, by multiple threads. Note that Haskell's *monads* were invented to deal with single-threaded global state, and that multithreaded access to global state inflicts additional violence on the functional model.

2. Multithreaded procedural languages often use synchronization primitives such as locks, atomic operations, and transactions, which inflict added violence upon the functional model.

3. Procedural languages can *alias* function arguments, for example, by passing a pointer to the same structure via two different arguments to the same invocation of a given function. This can result in the function unknowingly updating that structure via two different (and possibly overlapping) code sequences, which greatly complicates analysis.

Of course, given the importance of global state, synchronization primitives, and aliasing, clever functional-programming experts have proposed any number of attempts to reconcile the function programming model to them, monads being but one case in point.

Another approach is to compile the parallel procedural program into a functional program, then to use functional-programming tools to analyze the result. But it is possible to do much better than this, given that any real computation is a large finite-state machine with finite input that runs for a finite time interval. This means that any real program can be transformed into an expression, possibly albeit an impractically large one [DHK12].

However, a number of the low-level kernels of parallel algorithms transform into expressions that are small enough to fit easily into the memories of modern computers. If such an expression is coupled with an assertion, checking to see if the assertion would ever fire becomes a satisfiability problem. Even though satisfiability problems are NP-complete, they can often be solved in much less time than would be required to generate the full state space. In addition, the solution time appears to be only weakly dependent on the underlying memory model, so that algorithms running on weakly ordered systems can also be checked [AKT13].

The general approach is to transform the program into single-static-assignment (SSA) form, so that each assignment to a variable creates a separate version of that variable. This applies to assignments from all the active threads, so that the resulting expression embodies all possible executions of the code in question. The addition of an assertion entails asking whether any combination of inputs and initial values can result in the assertion firing, which, as noted above, is exactly the satisfiability problem.

One possible objection is that it does not gracefully handle arbitrary looping constructs. However, in many cases, this can be handled by unrolling the loop a finite

number of times. In addition, perhaps some loops will also prove amenable to collapse via inductive methods.

Another possible objection is that spinlocks involve arbitrarily long loops, and any finite unrolling would fail to capture the full behavior of the spinlock. It turns out that this objection is easily overcome. Instead of modeling a full spinlock, model a trylock that attempts to obtain the lock, and aborts if it fails to immediately do so. The assertion must then be crafted so as to avoid firing in cases where a spinlock aborted due to the lock not being immediately available. Because the logic expression is independent of time, all possible concurrency behaviors will be captured via this approach.

A final objection is that this technique is unlikely to be able to handle a full-sized software artifact such as the millions of lines of code making up the Linux kernel. This is likely the case, but the fact remains that exhaustive validation of each of the much smaller parallel primitives within the Linux kernel would be quite valuable. And in fact the researchers spearheading this approach have applied it to non-trivial real-world code, including the Tree RCU implementation in the Linux kernel [LMKM16, KS17a].

It remains to be seen how widely applicable this technique is, but it is one of the more interesting innovations in the field of formal verification. Although it might well be that the functional-programming advocates are at long last correct in their assertion of the inevitable dominance of functional programming, it is clearly the case that this long-touted methodology is starting to see credible competition on its formal-verification home turf. There is therefore continued reason to doubt the inevitability of functional-programming dominance.

17.6 Summary

This chapter has taken a quick tour of a number of possible futures, including multicore, transactional memory, formal verification as a regression test, and concurrent functional programming. Any of these futures might come true, but it is more likely that, as in the past, the future will be far stranger than we can possibly imagine.

www.dbooks.org

History is the sum total of things that could have been avoided.

Konrad Adenauer

Chapter 18

Looking Forward and Back

You have arrived at the end of this book, well done! I hope that your journey was a pleasant but challenging and worthwhile one.

For your editor and contributors, this is the end of the journey to the Second Edition, but for those willing to join in, it is also the start of the journey to the Third Edition. Either way, it is good to recap this past journey.

Chapter 1 covered what this book is about, along with some alternatives for those interested in something other than low-level parallel programming.

Chapter 2 covered parallel-programming challenges and high-level approaches for addressing them. It also touched on ways of avoiding these challenges while nevertheless still gaining most of the benefits of parallelism.

Chapter 3 gave a high-level overview of multicore hardware, especially aspects that pose challenges for concurrent software. This chapter puts the blame for these challenges where it belongs, very much on the laws of physics and rather less on intransigent hardware architects and designers. However, there might be some things that hardware architects and engineers can do, and this chapter discusses a few of them. In the meantime, software architects and engineers must meet these challenges, as discussed in the rest of the book.

Chapter 4 gave a quick overview of the tools of the low-level concurrency trade. Chapter 5 then demonstrated use of those tools—and, more importantly, use of parallel-programming design techniques—on the simple but surprisingly challenging task of concurrent counting. So challenging, in fact, that a number of concurrent counting algorithms are in common use, each specialized for a different use case.

Chapter 6 dug more deeply into the most important parallel-programming design technique, namely partitioning the problem at the highest possible level. This chapter also overviewed a number of points in this design space.

Chapter 7 expounded on that parallel-programming workhorse (and villain), locking. This chapter covered a number of types of locking and presented some engineering solutions to many well-known and aggressively advertised shortcomings of locking.

Chapter 8 discussed the uses of data ownership, where synchronization is supplied by the association of a given data item with a specific thread. Where it applies, this approach combines excellent performance and scalability with profound simplicity.

Chapter 9 showed how a little procrastination can greatly improve performance and scalabilty, while in a surprisingly large number of cases also simplifying the code. A number of the mechanisms presented in this chapter take advantage of the ability of CPU caches to replicate read-only data, thus sidestepping the laws of physics that cruelly limit the speed of light and the smallness of atoms. Chapter 10 looked at concurrent data structures, with emphasis on hash tables, which have a long and honorable history in parallel programs.

Chapter 11 dug into code-review and testing methods, and Chapter 12 overviewed formal verification. Whichever side of the formal-verification/testing divide you might be on, if code has not been thoroughly validated, it does not work. And that goes at least double for concurrent code.

Chapter 13 presented a number of situations where combining concurrency mechanisms with each other or with other design tricks can greatly ease parallel programmers' lives. Chapter 14 looked at advanced synchronization methods, including lockless programming, non-blocking synchronization, and parallel real-time computing. Chapter 15 dug into the critically important topic of memory ordering, presenting techniques and tools to help you not only solve memory-ordering problems, but also to avoid them completely. Chapter 16 presented a brief overview of the surprisingly important topic of ease of use.

www.dbooks.org

Last, but definitely not least, Chapter 17 expounded on a number of conflicting visions of the future, including CPU-technology trends, transactional memory, hardware transactional memory, use of formal verification in regression testing, and the long-standing prediction that the future of parallel programming belongs to functional-programming languages.

But now that we have recapped the contents of this Second Edition, how did this book get started?

Paul's parallel-programming journey started in earnest in 1990, when he joined Sequent Computer Systems, Inc. Sequent used an apprenticeship-like program in which newly hired engineers were placed in cubicles surrounded by experienced engineers, who mentored them, reviewed their code, and gave copious quantities of advice on a variety of topics. The result was that these newly hired engineers became productive parallel programmers within two or three months, and several of them were doing ground-breaking work within a couple of years.

Sequent understood that its ability to quickly train new engineers in the mysteries of parallelism was unusual, so it produced a slim volume that crystalized the company's parallel-programming wisdom [Seq88], which joined a pair of groundbreaking papers that had been written a few years earlier [BK85, Inm85]. People already steeped in these mysteries saluted this book and these papers, but novices were usually unable to benefit much from them, invariably making highly creative and quite destructive errors that were not explicitly prohibited by either the book or the papers.[1] This situation of course caused Paul to start thinking in terms of writing an improved book, but his efforts during this time were limited to internal training materials and to published papers.

By the time Sequent was acquired by IBM in 1999, many of the world's largest database instances ran on Sequent hardware. But times change, and by 2001 many of Sequent's parallel programmers had shifted their focus to the Linux kernel. After some initial reluctance, the Linux kernel community embraced concurrency both enthusiastically and effectively [BWCM+10, McK12a], with many excellent innovations and improvements from throughout the community. The thought of writing a book occurred to Paul from time to time, but life was flowing fast, so he made no progress on this project.

In 2006, Paul was invited to a conference on Linux scalability, and was granted the privilege of asking the last question of panel of esteemed parallel-programming experts. Paul began his question by noting that in the 15 years from 1991 to 2006, the price of a parallel system had dropped from that of a house to that of a mid-range bicycle, and it was clear that there was much more room for additional dramatic price decreases over the next 15 years extending to the year 2021. He also noted that decreasing price should result in greater familiarity and faster progress in solving parallel-programming problems. This led to his question: "In the year 2021, why wouldn't parallel programming have become routine?"

The first panelist seemed quite disdainful of anyone who would ask such an absurd question, and quickly responded with a soundbite answer. To which Paul gave a soundbite response. They went back and forth for some time, for example, the sound-bite answer "Deadlock" and the sound-bite response "Lock dependency checker".

The panelist eventually ran out of soundbites, and thus improvised a final "People like you should be hit over the head with a hammer!"

Paul's response was of course "You will have to get in line for that!"

Paul turned his attention to the next panelist, who seemed torn between agreeing with the first panelist and not wishing to have to deal with Paul's series of responses. He therefore have a short non-committal speech. And so it went through the rest of the panel.

Until it was the turn of the last panelist, who was someone you might have heard of who goes by the name of Linus Torvalds. Linus noted that three years earlier (that is, 2003), the initial version of any concurrency-related patch was usually quite poor, having design flaws and many bugs. And even when it was cleaned up enough to be accepted, bugs still remained. Linus contrasted this with the then-current situation in 2006, in which he said that it was not unusual for the first version of a concurrency-related patch to be well-designed with few or even no bugs. He then suggested that *if* tools continued to improve, then *maybe* parallel programming would become routine by the year 2021.

The conference then concluded. Paul was not surprised to be given wide berth by many audience members, especially those who saw the world in the same way as the first panelist saw it. Paul was also not surprised that a few audience members thanked him for the question. However, he was quite surprised when one man came up to say "thank you" with tears streaming down his face, sobbing so hard that he could barely speak.

You see, this man had worked several years at Sequent, and thus very well understood parallel programming. Furthermore, he was currently assigned to a group whose

[1] "But why on earth would you do *that*???" "Well, why not?"

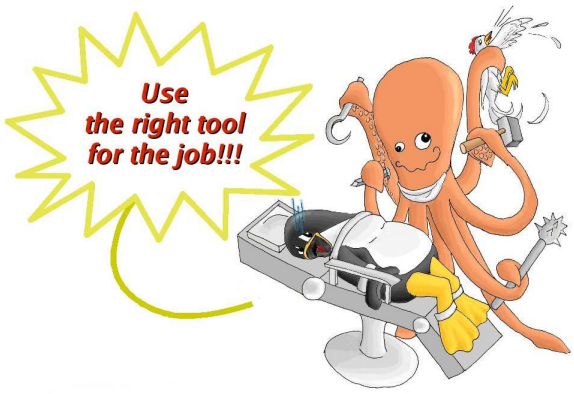


Figure 18.1: The Most Important Lesson

job it was to write parallel code. Which was not going well. You see, it wasn't that they had trouble understanding his explanations of parallel programming.

It was that they refused to listen to him *at all*.

In that moment, Paul went from "I should write a book some day" to "I will do whatever it takes to write this book". Paul is embarrassed to admit that he does not remember the man's name, if in fact he ever knew it.

This book is nevertheless for that man.

And this book is also for everyone else who would like to add low-level concurrency to their skillset. If you remember nothing else from this book, let it be the lesson of Figure 18.1.

For the rest of us, when someone tries to show us how to solve a pressing problem, perhaps we should do them the courtesy of at least listening!

www.dbooks.org

Ask me no questions, and I'll tell you no fibs.

"She Stoops to Conquer", Oliver Goldsmith

Appendix A

Important Questions

The following sections discuss some important questions relating to SMP programming. Each section also shows how to avoid worrying about the corresponding question, which can be extremely important if your goal is to simply get your SMP code working as quickly and painlessly as possible—which is an excellent goal, by the way!

Although the answers to these questions are often less intuitive than they would be in a single-threaded setting, with a bit of work, they are not that difficult to understand. If you managed to master recursion, there is nothing here that should pose an overwhelming challenge.

A.1 What Does "After" Mean?

"After" is an intuitive, but surprisingly difficult concept. An important non-intuitive issue is that code can be delayed at any point for any amount of time. Consider a producing and a consuming thread that communicate using a global struct with a timestamp "t" and integer fields "a", "b", and "c". The producer loops recording the current time (in seconds since 1970 in decimal), then updating the values of "a", "b", and "c", as shown in Listing A.1. The consumer code loops, also recording the current time, but also copying the producer's timestamp along with the fields "a", "b", and "c", as shown in Listing A.2. At the end of the run, the consumer outputs a list of anomalous recordings, e.g., where time has appeared to go backwards.

Quick Quiz A.1: What SMP coding errors can you see in these examples? See `time.c` for full code. ■

One might intuitively expect that the difference between the producer and consumer timestamps would be quite small, as it should not take much time for the producer to record the timestamps or the values. An excerpt of some sample output on a dual-core 1 GHz x86 is shown in Table A.1. Here, the "seq" column is the number of

Listing A.1: "After" Producer Function

```
1  /* WARNING: BUGGY CODE. */
2  void *producer(void *ignored)
3  {
4    int i = 0;
5
6    producer_ready = 1;
7    while (!goflag)
8      sched_yield();
9    while (goflag) {
10     ss.t = dgettimeofday();
11     ss.a = ss.c + 1;
12     ss.b = ss.a + 1;
13     ss.c = ss.b + 1;
14     i++;
15   }
16   printf("producer exiting: %d samples\n", i);
17   producer_done = 1;
18   return (NULL);
19 }
```

times through the loop, the "time" column is the time of the anomaly in seconds, the "delta" column is the number of seconds the consumer's timestamp follows that of the producer (where a negative value indicates that the consumer has collected its timestamp before the producer did), and the columns labelled "a", "b", and "c" show the amount that these variables increased since the prior snapshot collected by the consumer.

Table A.1: "After" Program Sample Output

seq	time (seconds)	delta	a	b	c
17563:	1152396.251585	(−16.928)	27	27	27
18004:	1152396.252581	(−12.875)	24	24	24
18163:	1152396.252955	(−19.073)	18	18	18
18765:	1152396.254449	(−148.773)	216	216	216
19863:	1152396.256960	(−6.914)	18	18	18
21644:	1152396.260959	(−5.960)	18	18	18
23408:	1152396.264957	(−20.027)	15	15	15

Why is time going backwards? The number in parentheses is the difference in microseconds, with a large number

Edition.2-rc11

www.dbooks.org

Listing A.2: "After" Consumer Function

```
1  /* WARNING: BUGGY CODE. */
2  void *consumer(void *ignored)
3  {
4    struct snapshot_consumer curssc;
5    int i = 0;
6    int j = 0;
7
8    consumer_ready = 1;
9    while (ss.t == 0.0) {
10     sched_yield();
11   }
12   while (goflag) {
13     curssc.tc = dgettimeofday();
14     curssc.t = ss.t;
15     curssc.a = ss.a;
16     curssc.b = ss.b;
17     curssc.c = ss.c;
18     curssc.sequence = curseq;
19     curssc.iserror = 0;
20     if ((curssc.t > curssc.tc) ||
21         modgreater(ssc[i].a, curssc.a) ||
22         modgreater(ssc[i].b, curssc.b) ||
23         modgreater(ssc[i].c, curssc.c) ||
24         modgreater(curssc.a, ssc[i].a + maxdelta) ||
25         modgreater(curssc.b, ssc[i].b + maxdelta) ||
26         modgreater(curssc.c, ssc[i].c + maxdelta)) {
27       i++;
28       curssc.iserror = 1;
29     } else if (ssc[i].iserror)
30       i++;
31     ssc[i] = curssc;
32     curseq++;
33     if (i + 1 >= NSNAPS)
34       break;
35   }
36   printf("consumer exited loop, collected %d items %d\n",
37          i, curseq);
38   if (ssc[0].iserror)
39     printf("0/%ld: %.6f %.6f (%.3f) %ld %ld %ld\n",
40            ssc[0].sequence,
41            ssc[j].t, ssc[j].tc,
42            (ssc[j].tc - ssc[j].t) * 1000000,
43            ssc[j].a, ssc[j].b, ssc[j].c);
44   for (j = 0; j <= i; j++)
45     if (ssc[j].iserror)
46       printf("%d/%ld: %.6f (%.3f) %ld %ld %ld\n",
47              j, ssc[j].sequence,
48              ssc[j].t, (ssc[j].tc - ssc[j].t) * 1000000,
49              ssc[j].a - ssc[j - 1].a,
50              ssc[j].b - ssc[j - 1].b,
51              ssc[j].c - ssc[j - 1].c);
52   consumer_done = 1;
53 }
```

exceeding 10 microseconds, and one exceeding even 100 microseconds! Please note that this CPU can potentially execute more than 100,000 instructions in that time.

One possible reason is given by the following sequence of events:

1. Consumer obtains timestamp (Listing A.2, line 13).
2. Consumer is preempted.
3. An arbitrary amount of time passes.
4. Producer obtains timestamp (Listing A.1, line 10).
5. Consumer starts running again, and picks up the producer's timestamp (Listing A.2, line 14).

In this scenario, the producer's timestamp might be an arbitrary amount of time after the consumer's timestamp.

How do you avoid agonizing over the meaning of "after" in your SMP code?

Simply use SMP primitives as designed.

In this example, the easiest fix is to use locking, for example, acquire a lock in the producer before line 10 in Listing A.1 and in the consumer before line 13 in Listing A.2. This lock must also be released after line 13 in Listing A.1 and after line 17 in Listing A.2. These locks cause the code segments in lines 10–13 of Listing A.1 and in lines 13–17 of Listing A.2 to *exclude* each other, in other words, to run atomically with respect to each other. This is represented in Figure A.1: the locking prevents any of the boxes of code from overlapping in time, so that the consumer's timestamp must be collected after the prior producer's timestamp. The segments of code in each box in this figure are termed "critical sections"; only one such critical section may be executing at a given time.

This addition of locking results in output as shown in Table A.2. Here there are no instances of time going backwards, instead, there are only cases with more than 1,000 counts difference between consecutive reads by the consumer.

Table A.2: Locked "After" Program Sample Output

seq	time (seconds)	delta	a	b	c
58597:	1156521.556296	(3.815)	1485	1485	1485
403927:	1156523.446636	(2.146)	2583	2583	2583

Quick Quiz A.2: How could there be such a large gap between successive consumer reads? See `timelocked.c` for full code. ■

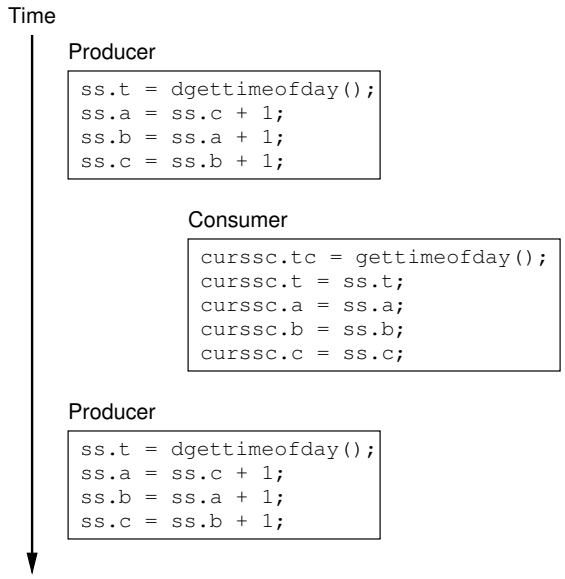


Figure A.1: Effect of Locking on Snapshot Collection

In summary, if you acquire an exclusive lock, you *know* that anything you do while holding that lock will appear to happen after anything done by any prior holder of that lock, at least give or take transactional lock elision (see Section 17.3.2.6). No need to worry about which CPU did or did not execute a memory barrier, no need to worry about the CPU or compiler reordering operations—life is simple. Of course, the fact that this locking prevents these two pieces of code from running concurrently might limit the program's ability to gain increased performance on multiprocessors, possibly resulting in a "safe but slow" situation. Chapter 6 describes ways of gaining performance and scalability in many situations.

However, in most cases, if you find yourself worrying about what happens before or after a given piece of code, you should take this as a hint to make better use of the standard primitives. Let these primitives do the worrying for you.

A.2 What is the Difference Between "Concurrent" and "Parallel"?

From a classic computing perspective, "concurrent" and "parallel" are clearly synonyms. However, this has not stopped many people from drawing distinctions between the two, and it turns out that these distinctions can be understood from a couple of different perspectives.

The first perspective treats "parallel" as an abbreviation for "data parallel", and treats "concurrent" as pretty much everything else. From this perspective, in parallel computing, each partition of the overall problem can proceed completely independently, with no communication with other partitions. In this case, little or no coordination among partitions is required. In contrast, concurrent computing might well have tight interdependencies, in the form of contended locks, transactions, or other synchronization mechanisms.

Quick Quiz A.3: Suppose a portion of a program uses RCU read-side primitives as its only synchronization mechanism. Is this parallelism or concurrency? ■

This of course begs the question of why such a distinction matters, which brings us to the second perspective, that of the underlying scheduler. Schedulers come in a wide range of complexities and capabilities, and as a rough rule of thumb, the more tightly and irregularly a set of parallel processes communicate, the higher the level of sophistication required from the scheduler. As such, parallel computing's avoidance of interdependencies means that parallel-computing programs run well on the least-capable schedulers. In fact, a pure parallel-computing program can run successfully after being arbitrarily subdivided and interleaved onto a uniprocessor.[1] In contrast, concurrent-computing programs might well require extreme subtlety on the part of the scheduler.

One could argue that we should simply demand a reasonable level of competence from the scheduler, so that we could simply ignore any distinctions between parallelism and concurrency. Although this is often a good strategy, there are important situations where efficiency, performance, and scalability concerns sharply limit the level of competence that the scheduler can reasonably offer. One important example is when the scheduler is implemented in hardware, as it often is in SIMD units or GPGPUs. Another example is a workload where the units of work are quite short, so that even a software-based scheduler must make hard choices between subtlety on the one hand and efficiency on the other.

Now, this second perspective can be thought of as making the workload match the available scheduler, with parallel workloads able to use simple schedulers and concurrent workloads requiring sophisticated schedulers.

Unfortunately, this perspective does not always align with the dependency-based distinction put forth by the

[1] Yes, this does mean that data-parallel-computing programs are best-suited for sequential execution. Why did you ask?

www.dbooks.org

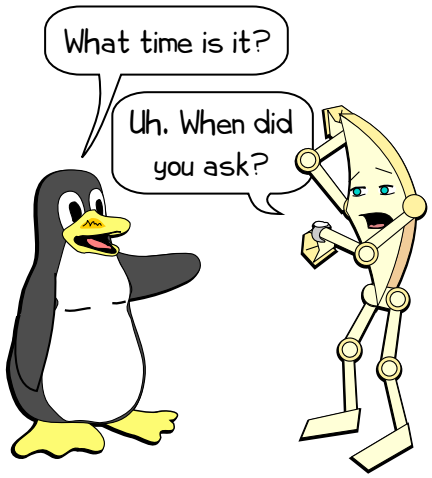


Figure A.2: What Time Is It?

first perspective. For example, a highly interdependent lock-based workload with one thread per CPU can make do with a trivial scheduler because no scheduler decisions are required. In fact, some workloads of this type can even be run one after another on a sequential machine. Therefore, such a workload would be labeled "concurrent" by the first perspective and "parallel" by many taking the second perspective.

Quick Quiz A.4: In what part of the second (scheduler-based) perspective would the lock-based single-thread-per-CPU workload be considered "concurrent"? ■

Which is just fine. No rule that humankind writes carries any weight against the objective universe, not even rules dividing multiprocessor programs into categories such as "concurrent" and "parallel".

This categorization failure does not mean such rules are useless, but rather that you should take on a suitably skeptical frame of mind when attempting to apply them to new situations. As always, use such rules where they apply and ignore them otherwise.

In fact, it is likely that new categories will arise in addition to parallel, concurrent, map-reduce, task-based, and so on. Some will stand the test of time, but good luck guessing which!

A.3 What Time Is It?

A key issue with timekeeping on multicore computer systems is illustrated by Figure A.2. One problem is that it takes time to read out the time. An instruction might read from a hardware clock, and might have to go off-core (or worse yet, off-socket) to complete this read operation. It might also be necessary to do some computation on the value read out, for example, to convert it to the desired format, to apply network time protocol (NTP) adjustments, and so on. So does the time eventually returned correspond to the beginning of the resulting time interval, the end, or somewhere in between?

Worse yet, the thread reading the time might be interrupted or preempted. Furthermore, there will likely be some computation between reading out the time and the actual use of the time that has been read out. Both of these possibilities further extend the interval of uncertainty.

One approach is to read the time twice, and take the arithmetic mean of the two readings, perhaps one on each side of the operation being timestamped. The difference between the two readings is then a measure of uncertainty of the time at which the intervening operation occurred.

Of course, in many cases, the exact time is not necessary. For example, when printing the time for the benefit of a human user, we can rely on slow human reflexes to render internal hardware and software delays irrelevant. Similarly, if a server needs to timestamp the response to a client, any time between the reception of the request and the transmission of the response will do equally well.

A.4 How Much Ordering?

How much ordering is enough?

Perhaps you have carefully constructed a strongly ordered concurrent system, only to find that it neither performs nor scales well. Or perhaps you threw caution to the wind, only to find that your brilliantly fast and scalable software is also unreliable. Is there a happy medium with both robust reliability on the one hand and powerful performance augmented by scintellating scalability on the other?

The answer, as is so often the case, is "it depends".

One approach is to construct a strongly ordered system, then examine its performance and scalablity. If these suffice, the system is good and sufficient, and no more need be done. Otherwise, undertake careful analysis (see Section 11.7) and attack each bottleneck until the system's performance is good and sufficient.

This approach can work very well, especially in contrast to the all-too-common approach of optimizing random components of the system in the hope of achieving significant system-wide benefits. However, starting with

strong ordering can also be quite wasteful, given that weakening ordering of the system's bottleneck can require that large portions of the rest of the system be redesigned and rewritten to accommodate the weakening. Worse yet, eliminating one bottleneck often exposes another, which in turn needs to be weakened and which in turn can result in wholesale redesigns and rewrites of other parts of the system. Perhaps even worse is the approach, also common, of starting with a fast but unreliable system and then playing whack-a-mole with an endless succession of concurrency bugs, though in the latter case, Chapters 11 and 12 are always there for you.

It would be better to have design-time tools to determine which portions of the system could use weak ordering, and at the same time, which portions actually benefit from weak ordering. These tasks are taken up by the following sections.

A.4.1 Where is the Defining Data?

One way to do this is to keep firmly in mind that the region of consistency engendered by strong ordering cannot extend out past the boundaries of the system.[2] Portions of the system whose role is to track the state of the outside world can usually feature weak ordering, given that speed-of-light delays will force the within-system state to lag that of the outside world. There is often no point in incurring large overheads to force a consistent view of data that is inherently out of date. In these cases, the methods of Chapter 9 can be quite helpful, as can some of the data structures described in Chapter 10.

Nevertheless, it is wise to adopt some meaningful semantics that are visible to those accessing the data, for example, a given function's return value might be:

1. Some value between the conceptual value at the time of the call to the function and the conceptual value at the time of the return from that function. For example, see the statistical counters discussed in Section 5.2, keeping in mind that such counters are normally monotonic, at least between consecutive overflows.

2. The actual value at some time between the call to and the return from that function. For example, see the single-variable atomic counter shown in Listing 5.2.

3. If the values used by that function remain unchanged during the time between that function's call and return, the expected value, otherwise some approximation to the expected value. Precise specification of the bounds on the approximation can be quite challenging. For example, consider a function combining values from different elements of an RCU-protected linked data structure, as described in Section 10.3.

In short, weaker ordering usually entails weaker consistency, and you should be able to give some sort of promise to your users as to how this weakening affects them. At the same time, unless the caller holds a lock across both the function call and the use of any values computed by that function, even fully ordered implementations normally cannot do any better than the semantics given by the options above.

Quick Quiz A.5: But if fully ordered implementations cannot offer stronger guarantees that the better performing and more scalable weakly ordered imptementations, why bother with full ordering? ■

Some might argue that useful computing deals only with the outside world, and therefore that all computing can use weak ordering. Such arguments are incorrect. For example, the value of your bank account is defined within your bank's computers, and people often prefer exact computations involving their account balances, especially those who might suspect that any such approximations would be in the bank's favor.

In short, although data tracking external state can be an attractive candidate for weakly ordered access, please think carefully about exactly what is being tracked and what is doing the tracking.

A.4.2 Consistent Data Used Consistently?

Another hint that weakening is safe can appear in the guise of data that is computed while holding a lock, but then used after the lock is released. The computed result clearly becomes at best an approximation as soon as the lock is released, which suggests computing an approximate result in the first place, possibly permitting use of weaker ordering. To this end, Chapter 5 covers numerous approximate methods for counting.

Great care is required, however. Is the use of data following lock release a hint that weak-ordering optimizations might be helpful? Or is instead a bug in which the lock was released too soon?

[2] Which might well be a distributed system.

www.dbooks.org

A.4.3 Is the Problem Partitionable?

Suppose that the system holds the defining instance of the data, or that using a computed value past lock release proved to be a bug. What then?

One approach is to partition the system, as discussed in Chapter 6. Partitioning can provide excellent scalability and in its more extreme form, per-CPU performance rivaling that of a sequential program, as discussed in Chapter 8. Partial partitioning is often mediated by locking, which is the subject of Chapter 7.

A.4.4 None of the Above?

The previous sections described the easier ways to gain performance and scalability, sometimes using weaker ordering and sometimes not. But the plain fact is that multicore systems are under no compunction to make life easy. But perhaps the advanced topics covered in Chapters 14 and 15 will prove helpful.

But please proceed with care, as it is all too easy to destabilize your codebase optimizing non-bottlenecks. Once again, Section 11.7 can help. It might also be worth your time to review other portions of this book, as it contains much information on handling a number of tricky situations.

The only difference between men and boys is the price of their toys.

M. Hébert

Appendix B

"Toy" RCU Implementations

The toy RCU implementations in this appendix are designed not for high performance, practicality, or any kind of production use,[1] but rather for clarity. Nevertheless, you will need a thorough understanding of Chapters 2, 3, 4, 6, and 9 for even these toy RCU implementations to be easily understandable.

This appendix provides a series of RCU implementations in order of increasing sophistication, from the viewpoint of solving the existence-guarantee problem. Appendix B.1 presents a rudimentary RCU implementation based on simple locking, while Appendices B.2 through B.9 present a series of simple RCU implementations based on locking, reference counters, and free-running counters. Finally, Appendix B.10 provides a summary and a list of desirable RCU properties.

B.1 Lock-Based RCU

Listing B.1: Lock-Based RCU Implementation

```
1  static void rcu_read_lock(void)
2  {
3    spin_lock(&rcu_gp_lock);
4  }
5
6  static void rcu_read_unlock(void)
7  {
8    spin_unlock(&rcu_gp_lock);
9  }
10
11 void synchronize_rcu(void)
12 {
13   spin_lock(&rcu_gp_lock);
14   spin_unlock(&rcu_gp_lock);
15 }
```

Perhaps the simplest RCU implementation leverages locking, as shown in Listing B.1 (`rcu_lock.h` and `rcu_lock.c`). In this implementation, `rcu_read_lock()` acquires a global spinlock, `rcu_read_unlock()` releases it, and `synchronize_rcu()` acquires it then immediately releases it.

Because `synchronize_rcu()` does not return until it has acquired (and released) the lock, it cannot return until all prior RCU read-side critical sections have completed, thus faithfully implementing RCU semantics. Of course, only one RCU reader may be in its read-side critical section at a time, which almost entirely defeats the purpose of RCU. In addition, the lock operations in `rcu_read_lock()` and `rcu_read_unlock()` are extremely heavyweight, with read-side overhead ranging from about 100 nanoseconds on a single POWER5 CPU up to more than 17 *microseconds* on a 64-CPU system. Worse yet, these same lock operations permit `rcu_read_lock()` to participate in deadlock cycles. Furthermore, in absence of recursive locks, RCU read-side critical sections cannot be nested, and, finally, although concurrent RCU updates could in principle be satisfied by a common grace period, this implementation serializes grace periods, preventing grace-period sharing.

Quick Quiz B.1: Why wouldn't any deadlock in the RCU implementation in Listing B.1 also be a deadlock in any other RCU implementation? ■

Quick Quiz B.2: Why not simply use reader-writer locks in the RCU implementation in Listing B.1 in order to allow RCU readers to proceed in parallel? ■

It is hard to imagine this implementation being useful in a production setting, though it does have the virtue of being implementable in almost any user-level application. Furthermore, similar implementations having one lock per CPU or using reader-writer locks have been used in production in the 2.4 Linux kernel.

[1] However, production-quality user-level RCU implementations are available [Des09b, DMS+12].

Edition.2-rc11

www.dbooks.org

Listing B.2: Per-Thread Lock-Based RCU Implementation

```
1  static void rcu_read_lock(void)
2  {
3    spin_lock(&__get_thread_var(rcu_gp_lock));
4  }
5
6  static void rcu_read_unlock(void)
7  {
8    spin_unlock(&__get_thread_var(rcu_gp_lock));
9  }
10
11 void synchronize_rcu(void)
12 {
13   int t;
14
15   for_each_running_thread(t) {
16     spin_lock(&per_thread(rcu_gp_lock, t));
17     spin_unlock(&per_thread(rcu_gp_lock, t));
18   }
19 }
```

A modified version of this one-lock-per-CPU approach, but instead using one lock per thread, is described in the next section.

B.2 Per-Thread Lock-Based RCU

Listing B.2 (`rcu_lock_percpu.h` and `rcu_lock_percpu.c`) shows an implementation based on one lock per thread. The `rcu_read_lock()` and `rcu_read_unlock()` functions acquire and release, respectively, the current thread's lock. The `synchronize_rcu()` function acquires and releases each thread's lock in turn. Therefore, all RCU read-side critical sections running when `synchronize_rcu()` starts must have completed before `synchronize_rcu()` can return.

This implementation does have the virtue of permitting concurrent RCU readers, and does avoid the deadlock condition that can arise with a single global lock. Furthermore, the read-side overhead, though high at roughly 140 nanoseconds, remains at about 140 nanoseconds regardless of the number of CPUs. However, the update-side overhead ranges from about 600 nanoseconds on a single POWER5 CPU up to more than 100 *microseconds* on 64 CPUs.

Quick Quiz B.3: Wouldn't it be cleaner to acquire all the locks, and then release them all in the loop from lines 15–18 of Listing B.2? After all, with this change, there would be a point in time when there were no readers, simplifying things greatly. ■

Quick Quiz B.4: Is the implementation shown in Listing B.2 free from deadlocks? Why or why not? ■

Listing B.3: RCU Implementation Using Single Global Reference Counter

```
1  atomic_t rcu_refcnt;
2
3  static void rcu_read_lock(void)
4  {
5    atomic_inc(&rcu_refcnt);
6    smp_mb();
7  }
8
9  static void rcu_read_unlock(void)
10 {
11   smp_mb();
12   atomic_dec(&rcu_refcnt);
13 }
14
15 void synchronize_rcu(void)
16 {
17   smp_mb();
18   while (atomic_read(&rcu_refcnt) != 0) {
19     poll(NULL, 0, 10);
20   }
21   smp_mb();
22 }
```

Quick Quiz B.5: Isn't one advantage of the RCU algorithm shown in Listing B.2 that it uses only primitives that are widely available, for example, in POSIX pthreads? ■

This approach could be useful in some situations, given that a similar approach was used in the Linux 2.4 kernel [MM00].

The counter-based RCU implementation described next overcomes some of the shortcomings of the lock-based implementation.

B.3 Simple Counter-Based RCU

A slightly more sophisticated RCU implementation is shown in Listing B.3 (`rcu_rcg.h` and `rcu_rcg.c`). This implementation makes use of a global reference counter `rcu_refcnt` defined on line 1. The `rcu_read_lock()` primitive atomically increments this counter, then executes a memory barrier to ensure that the RCU read-side critical section is ordered after the atomic increment. Similarly, `rcu_read_unlock()` executes a memory barrier to confine the RCU read-side critical section, then atomically decrements the counter. The `synchronize_rcu()` primitive spins waiting for the reference counter to reach zero, surrounded by memory barriers. The `poll()` on line 19 merely provides pure delay, and from a pure RCU-semantics point of view could be omitted. Again, once `synchronize_rcu()` returns, all prior RCU read-side critical sections are guaranteed to have completed.

In happy contrast to the lock-based implementation shown in Appendix B.1, this implementation allows par-

allel execution of RCU read-side critical sections. In happy contrast to the per-thread lock-based implementation shown in Appendix B.2, it also allows them to be nested. In addition, the `rcu_read_lock()` primitive cannot possibly participate in deadlock cycles, as it never spins nor blocks.

Quick Quiz B.6: But what if you hold a lock across a call to `synchronize_rcu()`, and then acquire that same lock within an RCU read-side critical section? ■

However, this implementation still has some serious shortcomings. First, the atomic operations in `rcu_read_lock()` and `rcu_read_unlock()` are still quite heavyweight, with read-side overhead ranging from about 100 nanoseconds on a single POWER5 CPU up to almost 40 *microseconds* on a 64-CPU system. This means that the RCU read-side critical sections have to be extremely long in order to get any real read-side parallelism. On the other hand, in the absence of readers, grace periods elapse in about 40 *nanoseconds*, many orders of magnitude faster than production-quality implementations in the Linux kernel.

Quick Quiz B.7: How can the grace period possibly elapse in 40 nanoseconds when `synchronize_rcu()` contains a 10-millisecond delay? ■

Second, if there are many concurrent `rcu_read_lock()` and `rcu_read_unlock()` operations, there will be extreme memory contention on `rcu_refcnt`, resulting in expensive cache misses. Both of these first two shortcomings largely defeat a major purpose of RCU, namely to provide low-overhead read-side synchronization primitives.

Finally, a large number of RCU readers with long read-side critical sections could prevent `synchronize_rcu()` from ever completing, as the global counter might never reach zero. This could result in starvation of RCU updates, which is of course unacceptable in production settings.

Quick Quiz B.8: Why not simply make `rcu_read_lock()` wait when a concurrent `synchronize_rcu()` has been waiting too long in the RCU implementation in Listing B.3? Wouldn't that prevent `synchronize_rcu()` from starving? ■

Therefore, it is still hard to imagine this implementation being useful in a production setting, though it has a bit more potential than the lock-based mechanism, for example, as an RCU implementation suitable for a high-stress debugging environment. The next section describes a variation on the reference-counting scheme that is more favorable to writers.

Listing B.4: RCU Global Reference-Count Pair Data

```
1 DEFINE_SPINLOCK(rcu_gp_lock);
2 atomic_t rcu_refcnt[2];
3 atomic_t rcu_idx;
4 DEFINE_PER_THREAD(int, rcu_nesting);
5 DEFINE_PER_THREAD(int, rcu_read_idx);
```

Listing B.5: RCU Read-Side Using Global Reference-Count Pair

```
1  static void rcu_read_lock(void)
2  {
3    int i;
4    int n;
5
6    n = __get_thread_var(rcu_nesting);
7    if (n == 0) {
8      i = atomic_read(&rcu_idx);
9      __get_thread_var(rcu_read_idx) = i;
10     atomic_inc(&rcu_refcnt[i]);
11   }
12   __get_thread_var(rcu_nesting) = n + 1;
13   smp_mb();
14 }
15
16 static void rcu_read_unlock(void)
17 {
18   int i;
19   int n;
20
21   smp_mb();
22   n = __get_thread_var(rcu_nesting);
23   if (n == 1) {
24     i = __get_thread_var(rcu_read_idx);
25     atomic_dec(&rcu_refcnt[i]);
26   }
27   __get_thread_var(rcu_nesting) = n - 1;
28 }
```

B.4 Starvation-Free Counter-Based RCU

Listing B.5 (`rcu_rcpg.h`) shows the read-side primitives of an RCU implementation that uses a pair of reference counters (`rcu_refcnt[]`), along with a global index that selects one counter out of the pair (`rcu_idx`), a per-thread nesting counter `rcu_nesting`, a per-thread snapshot of the global index (`rcu_read_idx`), and a global lock (`rcu_gp_lock`), which are themselves shown in Listing B.4.

Design It is the two-element `rcu_refcnt[]` array that provides the freedom from starvation. The key point is that `synchronize_rcu()` is only required to wait for pre-existing readers. If a new reader starts after a given instance of `synchronize_rcu()` has already

www.dbooks.org

begun execution, then that instance of synchronize_rcu() need not wait on that new reader. At any given time, when a given reader enters its RCU read-side critical section via rcu_read_lock(), it increments the element of the rcu_refcnt[] array indicated by the rcu_idx variable. When that same reader exits its RCU read-side critical section via rcu_read_unlock(), it decrements whichever element it incremented, ignoring any possible subsequent changes to the rcu_idx value.

This arrangement means that synchronize_rcu() can avoid starvation by complementing the value of rcu_idx, as in rcu_idx = !rcu_idx. Suppose that the old value of rcu_idx was zero, so that the new value is one. New readers that arrive after the complement operation will increment rcu_refcnt[1], while the old readers that previously incremented rcu_refcnt[0] will decrement rcu_refcnt[0] when they exit their RCU read-side critical sections. This means that the value of rcu_refcnt[0] will no longer be incremented, and thus will be monotonically decreasing.[2] This means that all that synchronize_rcu() need do is wait for the value of rcu_refcnt[0] to reach zero.

With the background, we are ready to look at the implementation of the actual primitives.

Implementation The rcu_read_lock() primitive atomically increments the member of the rcu_refcnt[] pair indexed by rcu_idx, and keeps a snapshot of this index in the per-thread variable rcu_read_idx. The rcu_read_unlock() primitive then atomically decrements whichever counter of the pair that the corresponding rcu_read_lock() incremented. However, because only one value of rcu_idx is remembered per thread, additional measures must be taken to permit nesting. These additional measures use the per-thread rcu_nesting variable to track nesting.

To make all this work, line 6 of rcu_read_lock() in Listing B.5 picks up the current thread's instance of rcu_nesting, and if line 7 finds that this is the outermost rcu_read_lock(), then lines 8–10 pick up the current value of rcu_idx, save it in this thread's instance of rcu_read_idx, and atomically increment the selected element of rcu_refcnt. Regardless of the value of rcu_nesting, line 12 increments it. Line 13 executes a memory barrier to ensure that the RCU read-side critical section does not bleed out before the rcu_read_lock() code.

Similarly, the rcu_read_unlock() function executes a memory barrier at line 21 to ensure that the RCU read-side critical section does not bleed out after the rcu_read_unlock() code. Line 22 picks up this thread's instance of rcu_nesting, and if line 23 finds that this is the outermost rcu_read_unlock(), then lines 24 and 25 pick up this thread's instance of rcu_read_idx (saved by the outermost rcu_read_lock()) and atomically decrements the selected element of rcu_refcnt. Regardless of the nesting level, line 27 decrements this thread's instance of rcu_nesting.

Listing B.6 (rcu_rcpg.c) shows the corresponding synchronize_rcu() implementation. Lines 6 and 19 acquire and release rcu_gp_lock in order to prevent more than one concurrent instance of synchronize_rcu(). Lines 7 and 8 pick up the value of rcu_idx and complement it, respectively, so that subsequent instances of rcu_read_lock() will use a different element of rcu_refcnt than did preceding instances. Lines 10–12 then wait for the prior element of rcu_refcnt to reach zero, with the memory barrier on line 9 ensuring that the check of rcu_refcnt is not reordered to precede the complementing of rcu_idx. Lines 13–18 repeat this process, and line 20 ensures that any subsequent reclamation operations are not reordered to precede the checking of rcu_refcnt.

Listing B.6: RCU Update Using Global Reference-Count Pair

```
1  void synchronize_rcu(void)
2  {
3    int i;
4
5    smp_mb();
6    spin_lock(&rcu_gp_lock);
7    i = atomic_read(&rcu_idx);
8    atomic_set(&rcu_idx, !i);
9    smp_mb();
10   while (atomic_read(&rcu_refcnt[i]) != 0) {
11     poll(NULL, 0, 10);
12   }
13   smp_mb();
14   atomic_set(&rcu_idx, i);
15   smp_mb();
16   while (atomic_read(&rcu_refcnt[!i]) != 0) {
17     poll(NULL, 0, 10);
18   }
19   spin_unlock(&rcu_gp_lock);
20   smp_mb();
21 }
```

Quick Quiz B.9: Why the memory barrier on line 5 of synchronize_rcu() in Listing B.6 given that there is a spin-lock acquisition immediately after? ■

[2] There is a race condition that this "monotonically decreasing" statement ignores. This race condition will be dealt with by the code for synchronize_rcu(). In the meantime, I suggest suspending disbelief.

Quick Quiz B.10: Why is the counter flipped twice in Listing B.6? Shouldn't a single flip-and-wait cycle be sufficient? ■

This implementation avoids the update-starvation issues that could occur in the single-counter implementation shown in Listing B.3.

Discussion There are still some serious shortcomings. First, the atomic operations in `rcu_read_lock()` and `rcu_read_unlock()` are still quite heavyweight. In fact, they are more complex than those of the single-counter variant shown in Listing B.3, with the read-side primitives consuming about 150 nanoseconds on a single POWER5 CPU and almost 40 *microseconds* on a 64-CPU system. The update-side `synchronize_rcu()` primitive is more costly as well, ranging from about 200 nanoseconds on a single POWER5 CPU to more than 40 *microseconds* on a 64-CPU system. This means that the RCU read-side critical sections have to be extremely long in order to get any real read-side parallelism.

Second, if there are many concurrent `rcu_read_lock()` and `rcu_read_unlock()` operations, there will be extreme memory contention on the `rcu_refcnt` elements, resulting in expensive cache misses. This further extends the RCU read-side critical-section duration required to provide parallel read-side access. These first two shortcomings defeat the purpose of RCU in most situations.

Third, the need to flip `rcu_idx` twice imposes substantial overhead on updates, especially if there are large numbers of threads.

Finally, despite the fact that concurrent RCU updates could in principle be satisfied by a common grace period, this implementation serializes grace periods, preventing grace-period sharing.

Quick Quiz B.11: Given that atomic increment and decrement are so expensive, why not just use non-atomic increment on line 10 and a non-atomic decrement on line 25 of Listing B.5? ■

Despite these shortcomings, one could imagine this variant of RCU being used on small tightly coupled multiprocessors, perhaps as a memory-conserving implementation that maintains API compatibility with more complex implementations. However, it would not likely scale well beyond a few CPUs.

The next section describes yet another variation on the reference-counting scheme that provides greatly improved read-side performance and scalability.

Listing B.7: RCU Per-Thread Reference-Count Pair Data

```
1 DEFINE_SPINLOCK(rcu_gp_lock);
2 DEFINE_PER_THREAD(int [2], rcu_refcnt);
3 atomic_t rcu_idx;
4 DEFINE_PER_THREAD(int, rcu_nesting);
5 DEFINE_PER_THREAD(int, rcu_read_idx);
```

Listing B.8: RCU Read-Side Using Per-Thread Reference-Count Pair

```
1  static void rcu_read_lock(void)
2  {
3    int i;
4    int n;
5
6    n = __get_thread_var(rcu_nesting);
7    if (n == 0) {
8      i = atomic_read(&rcu_idx);
9      __get_thread_var(rcu_read_idx) = i;
10     __get_thread_var(rcu_refcnt)[i]++;
11   }
12   __get_thread_var(rcu_nesting) = n + 1;
13   smp_mb();
14 }
15
16 static void rcu_read_unlock(void)
17 {
18   int i;
19   int n;
20
21   smp_mb();
22   n = __get_thread_var(rcu_nesting);
23   if (n == 1) {
24     i = __get_thread_var(rcu_read_idx);
25     __get_thread_var(rcu_refcnt)[i]--;
26   }
27   __get_thread_var(rcu_nesting) = n - 1;
28 }
```

B.5 Scalable Counter-Based RCU

Listing B.8 (`rcu_rcpl.h`) shows the read-side primitives of an RCU implementation that uses per-thread pairs of reference counters. This implementation is quite similar to that shown in Listing B.5, the only difference being that `rcu_refcnt` is now a per-thread array (as shown in Listing B.7). As with the algorithm in the previous section, use of this two-element array prevents readers from starving updaters. One benefit of per-thread `rcu_refcnt[]` array is that the `rcu_read_lock()` and `rcu_read_unlock()` primitives no longer perform atomic operations.

Quick Quiz B.12: Come off it! We can see the `atomic_read()` primitive in `rcu_read_lock()`!!! So why are you trying to pretend that `rcu_read_lock()` contains no atomic operations??? ■

Listing B.9 (`rcu_rcpl.c`) shows the implementation of `synchronize_rcu()`, along with a helper function named `flip_counter_and_wait()`. The

www.dbooks.org

Listing B.9: RCU Update Using Per-Thread Reference-Count Pair

```
1 static void flip_counter_and_wait(int i)
2 {
3   int t;
4
5   atomic_set(&rcu_idx, !i);
6   smp_mb();
7   for_each_thread(t) {
8     while (per_thread(rcu_refcnt, t)[i] != 0) {
9       poll(NULL, 0, 10);
10    }
11  }
12  smp_mb();
13 }
14
15 void synchronize_rcu(void)
16 {
17   int i;
18
19   smp_mb();
20   spin_lock(&rcu_gp_lock);
21   i = atomic_read(&rcu_idx);
22   flip_counter_and_wait(i);
23   flip_counter_and_wait(!i);
24   spin_unlock(&rcu_gp_lock);
25   smp_mb();
26 }
```

`synchronize_rcu()` function resembles that shown in Listing B.6, except that the repeated counter flip is replaced by a pair of calls on lines 22 and 23 to the new helper function.

The new `flip_counter_and_wait()` function updates the `rcu_idx` variable on line 5, executes a memory barrier on line 6, then lines 7–11 spin on each thread's prior `rcu_refcnt` element, waiting for it to go to zero. Once all such elements have gone to zero, it executes another memory barrier on line 12 and returns.

This RCU implementation imposes important new requirements on its software environment, namely, (1) that it be possible to declare per-thread variables, (2) that these per-thread variables be accessible from other threads, and (3) that it is possible to enumerate all threads. These requirements can be met in almost all software environments, but often result in fixed upper bounds on the number of threads. More-complex implementations might avoid such bounds, for example, by using expandable hash tables. Such implementations might dynamically track threads, for example, by adding them on their first call to `rcu_read_lock()`.

Quick Quiz B.13: Great, if we have N threads, we can have $2N$ ten-millisecond waits (one set per `flip_counter_and_wait()` invocation, and even that assumes that we wait only once for each thread. Don't we need the grace period to complete *much* more quickly? ■

Listing B.10: RCU Read-Side Using Per-Thread Reference-Count Pair and Shared Update Data

```
1 DEFINE_SPINLOCK(rcu_gp_lock);
2 DEFINE_PER_THREAD(int [2], rcu_refcnt);
3 long rcu_idx;
4 DEFINE_PER_THREAD(int, rcu_nesting);
5 DEFINE_PER_THREAD(int, rcu_read_idx);
```

This implementation still has several shortcomings. First, the need to flip `rcu_idx` twice imposes substantial overhead on updates, especially if there are large numbers of threads.

Second, `synchronize_rcu()` must now examine a number of variables that increases linearly with the number of threads, imposing substantial overhead on applications with large numbers of threads.

Third, as before, although concurrent RCU updates could in principle be satisfied by a common grace period, this implementation serializes grace periods, preventing grace-period sharing.

Finally, as noted in the text, the need for per-thread variables and for enumerating threads may be problematic in some software environments.

That said, the read-side primitives scale very nicely, requiring about 115 nanoseconds regardless of whether running on a single-CPU or a 64-CPU POWER5 system. As noted above, the `synchronize_rcu()` primitive does not scale, ranging in overhead from almost a microsecond on a single POWER5 CPU up to almost 200 microseconds on a 64-CPU system. This implementation could conceivably form the basis for a production-quality user-level RCU implementation.

The next section describes an algorithm permitting more efficient concurrent RCU updates.

B.6 Scalable Counter-Based RCU With Shared Grace Periods

Listing B.11 (`rcu_rcpls.h`) shows the read-side primitives for an RCU implementation using per-thread reference count pairs, as before, but permitting updates to share grace periods. The main difference from the earlier implementation shown in Listing B.8 is that `rcu_idx` is now a `long` that counts freely, so that line 8 of Listing B.11 must mask off the low-order bit. We also switched from using `atomic_read()` and `atomic_set()` to using `READ_ONCE()`. The data is also quite similar, as shown in Listing B.10, with `rcu_idx` now being a `long` instead of an `atomic_t`.

Listing B.11: RCU Read-Side Using Per-Thread Reference-Count Pair and Shared Update

```
1  static void rcu_read_lock(void)
2  {
3    int i;
4    int n;
5
6    n = __get_thread_var(rcu_nesting);
7    if (n == 0) {
8      i = READ_ONCE(rcu_idx) & 0x1;
9      __get_thread_var(rcu_read_idx) = i;
10     __get_thread_var(rcu_refcnt)[i]++;
11   }
12   __get_thread_var(rcu_nesting) = n + 1;
13   smp_mb();
14 }
15
16 static void rcu_read_unlock(void)
17 {
18   int i;
19   int n;
20
21   smp_mb();
22   n = __get_thread_var(rcu_nesting);
23   if (n == 1) {
24     i = __get_thread_var(rcu_read_idx);
25     __get_thread_var(rcu_refcnt)[i]--;
26   }
27   __get_thread_var(rcu_nesting) = n - 1;
28 }
```

Listing B.12: RCU Shared Update Using Per-Thread Reference-Count Pair

```
1  static void flip_counter_and_wait(int ctr)
2  {
3    int i;
4    int t;
5
6    WRITE_ONCE(rcu_idx, ctr + 1);
7    i = ctr & 0x1;
8    smp_mb();
9    for_each_thread(t) {
10     while (per_thread(rcu_refcnt, t)[i] != 0) {
11       poll(NULL, 0, 10);
12     }
13   }
14   smp_mb();
15 }
16
17 void synchronize_rcu(void)
18 {
19   int ctr;
20   int oldctr;
21
22   smp_mb();
23   oldctr = READ_ONCE(rcu_idx);
24   smp_mb();
25   spin_lock(&rcu_gp_lock);
26   ctr = READ_ONCE(rcu_idx);
27   if (ctr - oldctr >= 3) {
28     spin_unlock(&rcu_gp_lock);
29     smp_mb();
30     return;
31   }
32   flip_counter_and_wait(ctr);
33   if (ctr - oldctr < 2)
34     flip_counter_and_wait(ctr + 1);
35   spin_unlock(&rcu_gp_lock);
36   smp_mb();
37 }
```

Listing B.12 (`rcu_rcpls.c`) shows the implementation of `synchronize_rcu()` and its helper function `flip_counter_and_wait()`. These are similar to those in Listing B.9. The differences in `flip_counter_and_wait()` include:

1. Line 6 uses `WRITE_ONCE()` instead of `atomic_set()`, and increments rather than complementing.
2. A new line 7 masks the counter down to its bottom bit.

The changes to `synchronize_rcu()` are more pervasive:

1. There is a new `oldctr` local variable that captures the pre-lock-acquisition value of `rcu_idx` on line 20.
2. Line 23 uses `READ_ONCE()` instead of `atomic_read()`.
3. Lines 27–30 check to see if at least three counter flips were performed by other threads while the lock was being acquired, and, if so, releases the lock, does a memory barrier, and returns. In this case, there were two full waits for the counters to go to zero, so those other threads already did all the required work.
4. At lines 33–34, `flip_counter_and_wait()` is only invoked a second time if there were fewer than two counter flips while the lock was being acquired. On the other hand, if there were two counter flips, some other thread did one full wait for all the counters to go to zero, so only one more is required.

With this approach, if an arbitrarily large number of threads invoke `synchronize_rcu()` concurrently, with one CPU for each thread, there will be a total of only three waits for counters to go to zero.

Despite the improvements, this implementation of RCU still has a few shortcomings. First, as before, the need to flip `rcu_idx` twice imposes substantial overhead on updates, especially if there are large numbers of threads.

Second, each updater still acquires `rcu_gp_lock`, even if there is no work to be done. This can result in a severe scalability limitation if there are large numbers of concurrent updates. There are ways of avoiding this, as was done in a production-quality real-time implementation of RCU for the Linux kernel [McK07a].

Third, this implementation requires per-thread variables and the ability to enumerate threads, which again can be problematic in some software environments.

www.dbooks.org

Finally, on 32-bit machines, a given update thread might be preempted long enough for the rcu_idx counter to overflow. This could cause such a thread to force an unnecessary pair of counter flips. However, even if each grace period took only one microsecond, the offending thread would need to be preempted for more than an hour, in which case an extra pair of counter flips is likely the least of your worries.

As with the implementation described in Appendix B.3, the read-side primitives scale extremely well, incurring roughly 115 nanoseconds of overhead regardless of the number of CPUs. The synchronize_rcu() primitive is still expensive, ranging from about one microsecond up to about 16 microseconds. This is nevertheless much cheaper than the roughly 200 microseconds incurred by the implementation in Appendix B.5. So, despite its shortcomings, one could imagine this RCU implementation being used in production in real-life applications.

Quick Quiz B.14: All of these toy RCU implementations have either atomic operations in rcu_read_lock() and rcu_read_unlock(), or synchronize_rcu() overhead that increases linearly with the number of threads. Under what circumstances could an RCU implementation enjoy lightweight implementations for all three of these primitives, all having deterministic ($O(1)$) overheads and latencies? ■

Referring back to Listing B.11, we see that there is one global-variable access and no fewer than four accesses to thread-local variables. Given the relatively high cost of thread-local accesses on systems implementing POSIX threads, it is tempting to collapse the three thread-local variables into a single structure, permitting rcu_read_lock() and rcu_read_unlock() to access their thread-local data with a single thread-local-storage access. However, an even better approach would be to reduce the number of thread-local accesses to one, as is done in the next section.

B.7 RCU Based on Free-Running Counter

Listing B.14 (rcu.h and rcu.c) shows an RCU implementation based on a single global free-running counter that takes on only even-numbered values, with data shown in Listing B.13.

The resulting rcu_read_lock() implementation is extremely straightforward. Lines 3 and 4 simply add one to the global free-running rcu_gp_ctr variable and stores the resulting odd-numbered value into the rcu_reader_gp per-thread variable. Line 5 executes a memory barrier to prevent the content of the subsequent RCU read-side critical section from "leaking out".

Listing B.13: Data for Free-Running Counter Using RCU

```
1 DEFINE_SPINLOCK(rcu_gp_lock);
2 long rcu_gp_ctr = 0;
3 DEFINE_PER_THREAD(long, rcu_reader_gp);
4 DEFINE_PER_THREAD(long, rcu_reader_gp_snap);
```

Listing B.14: Free-Running Counter Using RCU

```
1  static inline void rcu_read_lock(void)
2  {
3    __get_thread_var(rcu_reader_gp) =
4      READ_ONCE(rcu_gp_ctr) + 1;
5    smp_mb();
6  }
7
8  static inline void rcu_read_unlock(void)
9  {
10   smp_mb();
11   __get_thread_var(rcu_reader_gp) =
12     READ_ONCE(rcu_gp_ctr);
13 }
14
15 void synchronize_rcu(void)
16 {
17   int t;
18
19   smp_mb();
20   spin_lock(&rcu_gp_lock);
21   WRITE_ONCE(rcu_gp_ctr, rcu_gp_ctr + 2);
22   smp_mb();
23   for_each_thread(t) {
24     while ((per_thread(rcu_reader_gp, t) & 0x1) &&
25            ((per_thread(rcu_reader_gp, t) -
26              rcu_gp_ctr) < 0)) {
27       poll(NULL, 0, 10);
28     }
29   }
30   spin_unlock(&rcu_gp_lock);
31   smp_mb();
32 }
```

The rcu_read_unlock() implementation is similar. Line 10 executes a memory barrier, again to prevent the prior RCU read-side critical section from "leaking out". Lines 11 and 12 then copy the rcu_gp_ctr global variable to the rcu_reader_gp per-thread variable, leaving this per-thread variable with an even-numbered value so that a concurrent instance of synchronize_rcu() will know to ignore it.

Quick Quiz B.15: If any even value is sufficient to tell synchronize_rcu() to ignore a given task, why don't lines 11 and 12 of Listing B.14 simply assign zero to rcu_reader_gp? ■

Thus, synchronize_rcu() could wait for all of the per-thread rcu_reader_gp variables to take on even-numbered values. However, it is possible to do much better

than that because synchronize_rcu() need only wait on *pre-existing* RCU read-side critical sections. Line 19 executes a memory barrier to prevent prior manipulations of RCU-protected data structures from being reordered (by either the CPU or the compiler) to follow the increment on line 21. Line 20 acquires the rcu_gp_lock (and line 30 releases it) in order to prevent multiple synchronize_rcu() instances from running concurrently. Line 21 then increments the global rcu_gp_ctr variable by two, so that all pre-existing RCU read-side critical sections will have corresponding per-thread rcu_reader_gp variables with values less than that of rcu_gp_ctr, modulo the machine's word size. Recall also that threads with even-numbered values of rcu_reader_gp are not in an RCU read-side critical section, so that lines 23–29 scan the rcu_reader_gp values until they all are either even (line 24) or are greater than the global rcu_gp_ctr (lines 25–26). Line 27 blocks for a short period of time to wait for a pre-existing RCU read-side critical section, but this can be replaced with a spin-loop if grace-period latency is of the essence. Finally, the memory barrier at line 31 ensures that any subsequent destruction will not be reordered into the preceding loop.

Quick Quiz B.16: Why are the memory barriers on lines 19 and 31 of Listing B.14 needed? Aren't the memory barriers inherent in the locking primitives on lines 20 and 30 sufficient? ■

This approach achieves much better read-side performance, incurring roughly 63 nanoseconds of overhead regardless of the number of POWER5 CPUs. Updates incur more overhead, ranging from about 500 nanoseconds on a single POWER5 CPU to more than 100 *microseconds* on 64 such CPUs.

Quick Quiz B.17: Couldn't the update-side batching optimization described in Appendix B.6 be applied to the implementation shown in Listing B.14? ■

This implementation suffers from some serious shortcomings in addition to the high update-side overhead noted earlier. First, it is no longer permissible to nest RCU read-side critical sections, a topic that is taken up in the next section. Second, if a reader is preempted at line 3 of Listing B.14 after fetching from rcu_gp_ctr but before storing to rcu_reader_gp, and if the rcu_gp_ctr counter then runs through more than half but less than all of its possible values, then synchronize_rcu() will ignore the subsequent RCU read-side critical section. Third and finally, this implementation requires that the enclosing software environment be able to enumerate threads and maintain per-thread variables.

Listing B.15: Data for Nestable RCU Using a Free-Running Counter

```
1 DEFINE_SPINLOCK(rcu_gp_lock);
2 #define RCU_GP_CTR_SHIFT 7
3 #define RCU_GP_CTR_BOTTOM_BIT (1 << RCU_GP_CTR_SHIFT)
4 #define RCU_GP_CTR_NEST_MASK (RCU_GP_CTR_BOTTOM_BIT - 1)
5 #define MAX_GP_ADV_DISTANCE (RCU_GP_CTR_NEST_MASK << 8)
6 unsigned long rcu_gp_ctr = 0;
7 DEFINE_PER_THREAD(unsigned long, rcu_reader_gp);
```

Quick Quiz B.18: Is the possibility of readers being preempted in lines 3–4 of Listing B.14 a real problem, in other words, is there a real sequence of events that could lead to failure? If not, why not? If so, what is the sequence of events, and how can the failure be addressed? ■

B.8 Nestable RCU Based on Free-Running Counter

Listing B.16 (rcu_nest.h and rcu_nest.c) shows an RCU implementation based on a single global free-running counter, but that permits nesting of RCU read-side critical sections. This nestability is accomplished by reserving the low-order bits of the global rcu_gp_ctr to count nesting, using the definitions shown in Listing B.15. This is a generalization of the scheme in Appendix B.7, which can be thought of as having a single low-order bit reserved for counting nesting depth. Two C-preprocessor macros are used to arrange this, RCU_GP_CTR_NEST_MASK and RCU_GP_CTR_BOTTOM_BIT. These are related: RCU_GP_CTR_NEST_MASK=RCU_GP_CTR_BOTTOM_BIT-1. The RCU_GP_CTR_BOTTOM_BIT macro contains a single bit that is positioned just above the bits reserved for counting nesting, and the RCU_GP_CTR_NEST_MASK has all one bits covering the region of rcu_gp_ctr used to count nesting. Obviously, these two C-preprocessor macros must reserve enough of the low-order bits of the counter to permit the maximum required nesting of RCU read-side critical sections, and this implementation reserves seven bits, for a maximum RCU read-side critical-section nesting depth of 127, which should be well in excess of that needed by most applications.

The resulting rcu_read_lock() implementation is still reasonably straightforward. Line 6 places a pointer to this thread's instance of rcu_reader_gp into the local variable rrgp, minimizing the number of expensive calls to the pthreads thread-local-state API. Line 7 records

www.dbooks.org

Listing B.16: Nestable RCU Using a Free-Running Counter

```
1  static void rcu_read_lock(void)
2  {
3    unsigned long tmp;
4    unsigned long *rrgp;
5
6    rrgp = &__get_thread_var(rcu_reader_gp);
7    tmp = *rrgp;
8    if ((tmp & RCU_GP_CTR_NEST_MASK) == 0)
9      tmp = READ_ONCE(rcu_gp_ctr);
10   tmp++;
11   WRITE_ONCE(*rrgp, tmp);
12   smp_mb();
13 }
14
15 static void rcu_read_unlock(void)
16 {
17   smp_mb();
18   __get_thread_var(rcu_reader_gp)--;
19 }
20
21 void synchronize_rcu(void)
22 {
23   int t;
24
25   smp_mb();
26   spin_lock(&rcu_gp_lock);
27   WRITE_ONCE(rcu_gp_ctr, rcu_gp_ctr +
28              RCU_GP_CTR_BOTTOM_BIT);
29   smp_mb();
30   for_each_thread(t) {
31     while (rcu_gp_ongoing(t) &&
32            ((READ_ONCE(per_thread(rcu_reader_gp, t)) -
33              rcu_gp_ctr) < 0)) {
34       poll(NULL, 0, 10);
35     }
36   }
37   spin_unlock(&rcu_gp_lock);
38   smp_mb();
39 }
```

the current value of `rcu_reader_gp` into another local variable `tmp`, and line 8 checks to see if the low-order bits are zero, which would indicate that this is the outermost `rcu_read_lock()`. If so, line 9 places the global `rcu_gp_ctr` into `tmp` because the current value previously fetched by line 7 is likely to be obsolete. In either case, line 10 increments the nesting depth, which you will recall is stored in the seven low-order bits of the counter. Line 11 stores the updated counter back into this thread's instance of `rcu_reader_gp`, and, finally, line 12 executes a memory barrier to prevent the RCU read-side critical section from bleeding out into the code preceding the call to `rcu_read_lock()`.

In other words, this implementation of `rcu_read_lock()` picks up a copy of the global `rcu_gp_ctr` unless the current invocation of `rcu_read_lock()` is nested within an RCU read-side critical section, in which case it instead fetches the contents of the current thread's instance of `rcu_reader_gp`. Either way, it increments whatever value it fetched in order to record an additional nesting level, and stores the result in the current thread's instance of `rcu_reader_gp`.

Interestingly enough, despite their `rcu_read_lock()` differences, the implementation of `rcu_read_unlock()` is broadly similar to that shown in Appendix B.7. Line 17 executes a memory barrier in order to prevent the RCU read-side critical section from bleeding out into code following the call to `rcu_read_unlock()`, and line 18 decrements this thread's instance of `rcu_reader_gp`, which has the effect of decrementing the nesting count contained in `rcu_reader_gp`'s low-order bits. Debugging versions of this primitive would check (before decrementing!) that these low-order bits were non-zero.

The implementation of `synchronize_rcu()` is quite similar to that shown in Appendix B.7. There are two differences. The first is that lines 27 and 28 adds `RCU_GP_CTR_BOTTOM_BIT` to the global `rcu_gp_ctr` instead of adding the constant "2", and the second is that the comparison on line 31 has been abstracted out to a separate function, where it checks the bit indicated by `RCU_GP_CTR_BOTTOM_BIT` instead of unconditionally checking the low-order bit.

This approach achieves read-side performance almost equal to that shown in Appendix B.7, incurring roughly 65 nanoseconds of overhead regardless of the number of POWER5 CPUs. Updates again incur more overhead, ranging from about 600 nanoseconds on a single POWER5 CPU to more than 100 *microseconds* on 64 such CPUs.

Quick Quiz B.19: Why not simply maintain a separate per-thread nesting-level variable, as was done in previous section, rather than having all this complicated bit manipulation? ■

This implementation suffers from the same shortcomings as does that of Appendix B.7, except that nesting of RCU read-side critical sections is now permitted. In addition, on 32-bit systems, this approach shortens the time required to overflow the global `rcu_gp_ctr` variable. The following section shows one way to greatly increase the time required for overflow to occur, while greatly reducing read-side overhead.

Quick Quiz B.20: Given the algorithm shown in Listing B.16, how could you double the time required to overflow the global `rcu_gp_ctr`? ■

Quick Quiz B.21: Again, given the algorithm shown in Listing B.16, is counter overflow fatal? Why or why not? If it is fatal, what can be done to fix it? ■

Listing B.17: Data for Quiescent-State-Based RCU

```
1 DEFINE_SPINLOCK(rcu_gp_lock);
2 long rcu_gp_ctr = 0;
3 DEFINE_PER_THREAD(long, rcu_reader_qs_gp);
```

B.9 RCU Based on Quiescent States

Listing B.18 (`rcu_qs.h`) shows the read-side primitives used to construct a user-level implementation of RCU based on quiescent states, with the data shown in Listing B.17. As can be seen from lines 1–7 in the listing, the `rcu_read_lock()` and `rcu_read_unlock()` primitives do nothing, and can in fact be expected to be inlined and optimized away, as they are in server builds of the Linux kernel. This is due to the fact that quiescent-state-based RCU implementations *approximate* the extents of RCU read-side critical sections using the aforementioned quiescent states. Each of these quiescent states contains a call to `rcu_quiescent_state()`, which is shown from lines 9–15 in the listing. Threads entering extended quiescent states (for example, when blocking) may instead call `rcu_thread_offline()` (lines 17–23) when entering an extended quiescent state and then call `rcu_thread_online()` (lines 25–28) when leaving it. As such, `rcu_thread_online()` is analogous to `rcu_read_lock()` and `rcu_thread_offline()` is analogous to `rcu_read_unlock()`. In addition, `rcu_quiescent_state()` can be thought of as a `rcu_thread_online()` immediately followed by a `rcu_thread_offline()`.[3] It is illegal to invoke `rcu_quiescent_state()`, `rcu_thread_offline()`, or `rcu_thread_online()` from an RCU read-side critical section.

In `rcu_quiescent_state()`, line 11 executes a memory barrier to prevent any code prior to the quiescent state (including possible RCU read-side critical sections) from being reordered into the quiescent state. Lines 12–13 pick up a copy of the global `rcu_gp_ctr`, using `READ_ONCE()` to ensure that the compiler does not employ any optimizations that would result in `rcu_gp_ctr` being fetched more than once, and then adds one to the value fetched and stores it into the per-thread `rcu_reader_qs_gp` variable, so that any concurrent instance of `synchronize_rcu()` will see an odd-numbered value, thus becoming aware that a new RCU read-side critical section has started. Instances of `synchronize_rcu()` that are waiting on older RCU read-side critical sections will thus know to ignore this new one. Finally, line 14 executes a memory barrier, which prevents subsequent code (including a possible RCU read-side critical section) from being re-ordered with the lines 12–13.

[3] Although the code in the listing is consistent with `rcu_quiescent_state()` being the same as `rcu_thread_online()` immediately followed by `rcu_thread_offline()`, this relationship is obscured by performance optimizations.

Listing B.18: Quiescent-State-Based RCU Read Side

```
1  static void rcu_read_lock(void)
2  {
3  }
4
5  static void rcu_read_unlock(void)
6  {
7  }
8
9  static void rcu_quiescent_state(void)
10 {
11   smp_mb();
12   __get_thread_var(rcu_reader_qs_gp) =
13     READ_ONCE(rcu_gp_ctr) + 1;
14   smp_mb();
15 }
16
17 static void rcu_thread_offline(void)
18 {
19   smp_mb();
20   __get_thread_var(rcu_reader_qs_gp) =
21     READ_ONCE(rcu_gp_ctr);
22   smp_mb();
23 }
24
25 static void rcu_thread_online(void)
26 {
27   rcu_quiescent_state();
28 }
```

Quick Quiz B.22: Doesn't the additional memory barrier shown on line 14 of Listing B.18 greatly increase the overhead of `rcu_quiescent_state`? ■

Some applications might use RCU only occasionally, but use it very heavily when they do use it. Such applications might choose to use `rcu_thread_online()` when starting to use RCU and `rcu_thread_offline()` when no longer using RCU. The time between a call to `rcu_thread_offline()` and a subsequent call to `rcu_thread_online()` is an extended quiescent state, so that RCU will not expect explicit quiescent states to be registered during this time.

The `rcu_thread_offline()` function simply sets the per-thread `rcu_reader_qs_gp` variable to the current value of `rcu_gp_ctr`, which has an even-numbered value. Any concurrent instances of `synchronize_rcu()` will thus know to ignore this thread.

Quick Quiz B.23: Why are the two memory barriers on lines 11 and 14 of Listing B.18 needed? ■

www.dbooks.org

Listing B.19: RCU Update Side Using Quiescent States

```
1  void synchronize_rcu(void)
2  {
3    int t;
4
5    smp_mb();
6    spin_lock(&rcu_gp_lock);
7    WRITE_ONCE(rcu_gp_ctr, rcu_gp_ctr + 2);
8    smp_mb();
9    for_each_thread(t) {
10     while (rcu_gp_ongoing(t) &&
11            ((per_thread(rcu_reader_qs_gp, t)
12              - rcu_gp_ctr) < 0)) {
13       poll(NULL, 0, 10);
14     }
15   }
16   spin_unlock(&rcu_gp_lock);
17   smp_mb();
18 }
```

The `rcu_thread_online()` function simply invokes `rcu_quiescent_state()`, thus marking the end of the extended quiescent state.

Listing B.19 (`rcu_qs.c`) shows the implementation of `synchronize_rcu()`, which is quite similar to that of the preceding sections.

This implementation has blazingly fast read-side primitives, with an `rcu_read_lock()`–`rcu_read_unlock()` round trip incurring an overhead of roughly 50 *picoseconds*. The `synchronize_rcu()` overhead ranges from about 600 nanoseconds on a single-CPU POWER5 system up to more than 100 microseconds on a 64-CPU system.

Quick Quiz B.24: To be sure, the clock frequencies of POWER systems in 2008 were quite high, but even a 5 GHz clock frequency is insufficient to allow loops to be executed in 50 picoseconds! What is going on here? ■

However, this implementation requires that each thread either invoke `rcu_quiescent_state()` periodically or to invoke `rcu_thread_offline()` for extended quiescent states. The need to invoke these functions periodically can make this implementation difficult to use in some situations, such as for certain types of library functions.

Quick Quiz B.25: Why would the fact that the code is in a library make any difference for how easy it is to use the RCU implementation shown in Listings B.18 and B.19? ■

Quick Quiz B.26: But what if you hold a lock across a call to `synchronize_rcu()`, and then acquire that same lock within an RCU read-side critical section? This should be a deadlock, but how can a primitive that generates absolutely no code possibly participate in a deadlock cycle? ■

In addition, this implementation does not permit concurrent calls to `synchronize_rcu()` to share grace periods. That said, one could easily imagine a production-quality RCU implementation based on this version of RCU.

B.10 Summary of Toy RCU Implementations

If you made it this far, congratulations! You should now have a much clearer understanding not only of RCU itself, but also of the requirements of enclosing software environments and applications. Those wishing an even deeper understanding are invited to read descriptions of production-quality RCU implementations [DMS+12, McK07a, McK08a, McK09a].

The preceding sections listed some desirable properties of the various RCU primitives. The following list is provided for easy reference for those wishing to create a new RCU implementation.

1. There must be read-side primitives (such as `rcu_read_lock()` and `rcu_read_unlock()`) and grace-period primitives (such as `synchronize_rcu()` and `call_rcu()`), such that any RCU read-side critical section in existence at the start of a grace period has completed by the end of the grace period.
2. RCU read-side primitives should have minimal overhead. In particular, expensive operations such as cache misses, atomic instructions, memory barriers, and branches should be avoided.
3. RCU read-side primitives should have $O(1)$ computational complexity to enable real-time use. (This implies that readers run concurrently with updaters.)
4. RCU read-side primitives should be usable in all contexts (in the Linux kernel, they are permitted everywhere except in the idle loop). An important special case is that RCU read-side primitives be usable within an RCU read-side critical section, in other words, that it be possible to nest RCU read-side critical sections.
5. RCU read-side primitives should be unconditional, with no failure returns. This property is extremely important, as failure checking increases complexity and complicates testing and validation.
6. Any operation other than a quiescent state (and thus a grace period) should be permitted in an RCU read-side critical section. In particular, irrevocable operations such as I/O should be permitted.

7. It should be possible to update an RCU-protected data structure while executing within an RCU read-side critical section.

8. Both RCU read-side and update-side primitives should be independent of memory allocator design and implementation, in other words, the same RCU implementation should be able to protect a given data structure regardless of how the data elements are allocated and freed.

9. RCU grace periods should not be blocked by threads that halt outside of RCU read-side critical sections. (But note that most quiescent-state-based implementations violate this desideratum.)

Quick Quiz B.27: Given that grace periods are prohibited within RCU read-side critical sections, how can an RCU data structure possibly be updated while in an RCU read-side critical section? ■

www.dbooks.org

Order! Order in the court!

Unknown

Appendix C

Why Memory Barriers?

So what possessed CPU designers to cause them to inflict memory barriers on poor unsuspecting SMP software designers?

In short, because reordering memory references allows much better performance, and so memory barriers are needed to force ordering in things like synchronization primitives whose correct operation depends on ordered memory references.

Getting a more detailed answer to this question requires a good understanding of how CPU caches work, and especially what is required to make caches really work well. The following sections:

1. present the structure of a cache,
2. describe how cache-coherency protocols ensure that CPUs agree on the value of each location in memory, and, finally,
3. outline how store buffers and invalidate queues help caches and cache-coherency protocols achieve high performance.

We will see that memory barriers are a necessary evil that is required to enable good performance and scalability, an evil that stems from the fact that CPUs are orders of magnitude faster than are both the interconnects between them and the memory they are attempting to access.

C.1 Cache Structure

Modern CPUs are much faster than are modern memory systems. A 2006 CPU might be capable of executing ten instructions per nanosecond, but will require many tens of nanoseconds to fetch a data item from main memory. This disparity in speed—more than two orders of magnitude—has resulted in the multi-megabyte caches found on modern CPUs. These caches are associated with the CPUs as shown in Figure C.1, and can typically be accessed in a few cycles.[1]

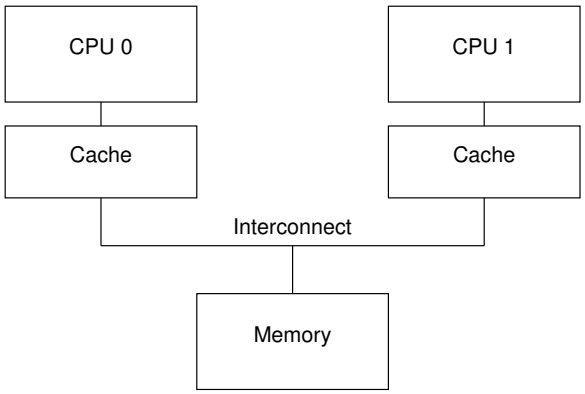


Figure C.1: Modern Computer System Cache Structure

Data flows among the CPUs' caches and memory in fixed-length blocks called "cache lines", which are normally a power of two in size, ranging from 16 to 256 bytes. When a given data item is first accessed by a given CPU, it will be absent from that CPU's cache, meaning that a "cache miss" (or, more specifically, a "startup" or "warmup" cache miss) has occurred. The cache miss means that the CPU will have to wait (or be "stalled") for hundreds of cycles while the item is fetched from memory. However, the item will be loaded into that CPU's cache, so that subsequent accesses will find it in the cache and therefore run at full speed.

[1] It is standard practice to use multiple levels of cache, with a small level-one cache close to the CPU with single-cycle access time, and a larger level-two cache with a longer access time, perhaps roughly ten clock cycles. Higher-performance CPUs often have three or even four levels of cache.

www.dbooks.org

	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x43210E00
0xF		

Figure C.2: CPU Cache Structure

After some time, the CPU's cache will fill, and subsequent misses will likely need to eject an item from the cache in order to make room for the newly fetched item. Such a cache miss is termed a "capacity miss", because it is caused by the cache's limited capacity. However, most caches can be forced to eject an old item to make room for a new item even when they are not yet full. This is due to the fact that large caches are implemented as hardware hash tables with fixed-size hash buckets (or "sets", as CPU designers call them) and no chaining, as shown in Figure C.2.

This cache has sixteen "sets" and two "ways" for a total of 32 "lines", each entry containing a single 256-byte "cache line", which is a 256-byte-aligned block of memory. This cache line size is a little on the large size, but makes the hexadecimal arithmetic much simpler. In hardware parlance, this is a two-way set-associative cache, and is analogous to a software hash table with sixteen buckets, where each bucket's hash chain is limited to at most two elements. The size (32 cache lines in this case) and the associativity (two in this case) are collectively called the cache's "geometry". Since this cache is implemented in hardware, the hash function is extremely simple: extract four bits from the memory address.

In Figure C.2, each box corresponds to a cache entry, which can contain a 256-byte cache line. However, a cache entry can be empty, as indicated by the empty boxes in the figure. The rest of the boxes are flagged with the memory address of the cache line that they contain. Since the cache lines must be 256-byte aligned, the low eight bits of each address are zero, and the choice of hardware hash function means that the next-higher four bits match the hash line number.

The situation depicted in the figure might arise if the program's code were located at address 0x43210E00 through 0x43210EFF, and this program accessed data sequentially from 0x12345000 through 0x12345EFF. Suppose that the program were now to access location 0x12345F00. This location hashes to line 0xF, and both ways of this line are empty, so the corresponding 256-byte line can be accommodated. If the program were to access location 0x1233000, which hashes to line 0x0, the corresponding 256-byte cache line can be accommodated in way 1. However, if the program were to access location 0x1233E00, which hashes to line 0xE, one of the existing lines must be ejected from the cache to make room for the new cache line. If this ejected line were accessed later, a cache miss would result. Such a cache miss is termed an "associativity miss".

Thus far, we have been considering only cases where a CPU reads a data item. What happens when it does a write? Because it is important that all CPUs agree on the value of a given data item, before a given CPU writes to that data item, it must first cause it to be removed, or "invalidated", from other CPUs' caches. Once this invalidation has completed, the CPU may safely modify the data item. If the data item was present in this CPU's cache, but was read-only, this process is termed a "write miss". Once a given CPU has completed invalidating a given data item from other CPUs' caches, that CPU may repeatedly write (and read) that data item.

Later, if one of the other CPUs attempts to access the data item, it will incur a cache miss, this time because the first CPU invalidated the item in order to write to it. This type of cache miss is termed a "communication miss", since it is usually due to several CPUs using the data items to communicate (for example, a lock is a data item that is used to communicate among CPUs using a mutual-exclusion algorithm).

Clearly, much care must be taken to ensure that all CPUs maintain a coherent view of the data. With all this fetching, invalidating, and writing, it is easy to imagine data being lost or (perhaps worse) different CPUs having conflicting values for the same data item in their respective caches. These problems are prevented by "cache-coherency protocols", described in the next section.

C.2 Cache-Coherence Protocols

Cache-coherency protocols manage cache-line states so as to prevent inconsistent or lost data. These protocols can be quite complex, with many tens of states,[2] but for our purposes we need only concern ourselves with the four-state MESI cache-coherence protocol.

C.2.1 MESI States

MESI stands for "modified", "exclusive", "shared", and "invalid", the four states a given cache line can take on using this protocol. Caches using this protocol therefore maintain a two-bit state "tag" on each cache line in addition to that line's physical address and data.

A line in the "modified" state has been subject to a recent memory store from the corresponding CPU, and the corresponding memory is guaranteed not to appear in any other CPU's cache. Cache lines in the "modified" state can thus be said to be "owned" by the CPU. Because this cache holds the only up-to-date copy of the data, this cache is ultimately responsible for either writing it back to memory or handing it off to some other cache, and must do so before reusing this line to hold other data.

The "exclusive" state is very similar to the "modified" state, the single exception being that the cache line has not yet been modified by the corresponding CPU, which in turn means that the copy of the cache line's data that resides in memory is up-to-date. However, since the CPU can store to this line at any time, without consulting other CPUs, a line in the "exclusive" state can still be said to be owned by the corresponding CPU. That said, because the corresponding value in memory is up to date, this cache can discard this data without writing it back to memory or handing it off to some other CPU.

A line in the "shared" state might be replicated in at least one other CPU's cache, so that this CPU is not permitted to store to the line without first consulting with other CPUs. As with the "exclusive" state, because the corresponding value in memory is up to date, this cache can discard this data without writing it back to memory or handing it off to some other CPU.

A line in the "invalid" state is empty, in other words, it holds no data. When new data enters the cache, it is placed into a cache line that was in the "invalid" state if possible. This approach is preferred because replacing a line in any other state could result in an expensive cache miss should the replaced line be referenced in the future.

Since all CPUs must maintain a coherent view of the data carried in the cache lines, the cache-coherence protocol provides messages that coordinate the movement of cache lines through the system.

C.2.2 MESI Protocol Messages

Many of the transitions described in the previous section require communication among the CPUs. If the CPUs are on a single shared bus, the following messages suffice:

Read:
The "read" message contains the physical address of the cache line to be read.

Read Response:
The "read response" message contains the data requested by an earlier "read" message. This "read response" message might be supplied either by memory or by one of the other caches. For example, if one of the caches has the desired data in "modified" state, that cache must supply the "read response" message.

Invalidate:
The "invalidate" message contains the physical address of the cache line to be invalidated. All other caches must remove the corresponding data from their caches and respond.

Invalidate Acknowledge:
A CPU receiving an "invalidate" message must respond with an "invalidate acknowledge" message after removing the specified data from its cache.

Read Invalidate:
The "read invalidate" message contains the physical address of the cache line to be read, while at the same time directing other caches to remove the data. Hence, it is a combination of a "read" and an "invalidate", as indicated by its name. A "read invalidate" message requires both a "read response" and a set of "invalidate acknowledge" messages in reply.

Writeback:
The "writeback" message contains both the address and the data to be written back to memory (and perhaps "snooped" into other CPUs' caches along the way). This message permits caches to eject lines in the "modified" state as needed to make room for other data.

[2] See Culler et al. [CSG99] pages 670 and 671 for the nine-state and 26-state diagrams for SGI Origin2000 and Sequent (now IBM) NUMA-Q, respectively. Both diagrams are significantly simpler than real life.

www.dbooks.org

Quick Quiz C.1: Where does a writeback message originate from and where does it go to? ■

Interestingly enough, a shared-memory multiprocessor system really is a message-passing computer under the covers. This means that clusters of SMP machines that use distributed shared memory are using message passing to implement shared memory at two different levels of the system architecture.

Quick Quiz C.2: What happens if two CPUs attempt to invalidate the same cache line concurrently? ■

Quick Quiz C.3: When an "invalidate" message appears in a large multiprocessor, every CPU must give an "invalidate acknowledge" response. Wouldn't the resulting "storm" of "invalidate acknowledge" responses totally saturate the system bus? ■

Quick Quiz C.4: If SMP machines are really using message passing anyway, why bother with SMP at all? ■

C.2.3 MESI State Diagram

A given cache line's state changes as protocol messages are sent and received, as shown in Figure C.3.

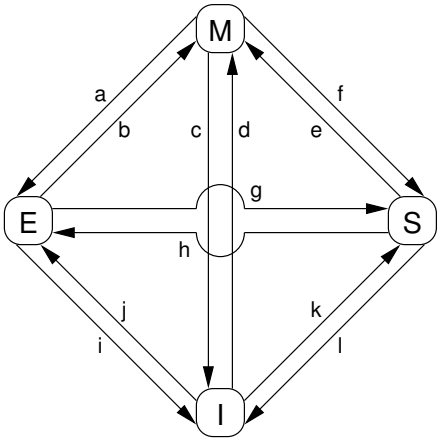


Figure C.3: MESI Cache-Coherency State Diagram

The transition arcs in this figure are as follows:

Transition (a):
A cache line is written back to memory, but the CPU retains it in its cache and further retains the right to modify it. This transition requires a "writeback" message.

Transition (b):
The CPU writes to the cache line that it already had exclusive access to. This transition does not require any messages to be sent or received.

Transition (c):
The CPU receives a "read invalidate" message for a cache line that it has modified. The CPU must invalidate its local copy, then respond with both a "read response" and an "invalidate acknowledge" message, both sending the data to the requesting CPU and indicating that it no longer has a local copy.

Transition (d):
The CPU does an atomic read-modify-write operation on a data item that was not present in its cache. It transmits a "read invalidate", receiving the data via a "read response". The CPU can complete the transition once it has also received a full set of "invalidate acknowledge" responses.

Transition (e):
The CPU does an atomic read-modify-write operation on a data item that was previously read-only in its cache. It must transmit "invalidate" messages, and must wait for a full set of "invalidate acknowledge" responses before completing the transition.

Transition (f):
Some other CPU reads the cache line, and it is supplied from this CPU's cache, which retains a read-only copy, possibly also writing it back to memory. This transition is initiated by the reception of a "read" message, and this CPU responds with a "read response" message containing the requested data.

Transition (g):
Some other CPU reads a data item in this cache line, and it is supplied either from this CPU's cache or from memory. In either case, this CPU retains a read-only copy. This transition is initiated by the reception of a "read" message, and this CPU responds with a "read response" message containing the requested data.

Transition (h):
This CPU realizes that it will soon need to write to some data item in this cache line, and thus transmits an "invalidate" message. The CPU cannot complete the transition until it receives a full set of "invalidate acknowledge" responses. Alternatively, all other CPUs eject this cache line from their caches via

"writeback" messages (presumably to make room for other cache lines), so that this CPU is the last CPU caching it.

Transition (i):
Some other CPU does an atomic read-modify-write operation on a data item in a cache line held only in this CPU's cache, so this CPU invalidates it from its cache. This transition is initiated by the reception of a "read invalidate" message, and this CPU responds with both a "read response" and an "invalidate acknowledge" message.

Transition (j):
This CPU does a store to a data item in a cache line that was not in its cache, and thus transmits a "read invalidate" message. The CPU cannot complete the transition until it receives the "read response" and a full set of "invalidate acknowledge" messages. The cache line will presumably transition to "modified" state via transition (b) as soon as the actual store completes.

Transition (k):
This CPU loads a data item in a cache line that was not in its cache. The CPU transmits a "read" message, and completes the transition upon receiving the corresponding "read response".

Transition (l):
Some other CPU does a store to a data item in this cache line, but holds this cache line in read-only state due to its being held in other CPUs' caches (such as the current CPU's cache). This transition is initiated by the reception of an "invalidate" message, and this CPU responds with an "invalidate acknowledge" message.

Quick Quiz C.5: How does the hardware handle the delayed transitions described above? ■

C.2.4 MESI Protocol Example

Let's now look at this from the perspective of a cache line's worth of data, initially residing in memory at address 0, as it travels through the various single-line direct-mapped caches in a four-CPU system. Table C.1 shows this flow of data, with the first column showing the sequence of operations, the second the CPU performing the operation, the third the operation being performed, the next four the state of each CPU's cache line (memory address followed by MESI state), and the final two columns whether the corresponding memory contents are up to date ("V") or not ("I").

Initially, the CPU cache lines in which the data would reside are in the "invalid" state, and the data is valid in memory. When CPU 0 loads the data at address 0, it enters the "shared" state in CPU 0's cache, and is still valid in memory. CPU 3 also loads the data at address 0, so that it is in the "shared" state in both CPUs' caches, and is still valid in memory. Next CPU 0 loads some other cache line (at address 8), which forces the data at address 0 out of its cache via an invalidation, replacing it with the data at address 8. CPU 2 now does a load from address 0, but this CPU realizes that it will soon need to store to it, and so it uses a "read invalidate" message in order to gain an exclusive copy, invalidating it from CPU 3's cache (though the copy in memory remains up to date). Next CPU 2 does its anticipated store, changing the state to "modified". The copy of the data in memory is now out of date. CPU 1 does an atomic increment, using a "read invalidate" to snoop the data from CPU 2's cache and invalidate it, so that the copy in CPU 1's cache is in the "modified" state (and the copy in memory remains out of date). Finally, CPU 1 reads the cache line at address 8, which uses a "writeback" message to push address 0's data back out to memory.

Note that we end with data in some of the CPU's caches.

Quick Quiz C.6: What sequence of operations would put the CPUs' caches all back into the "invalid" state? ■

C.3 Stores Result in Unnecessary Stalls

Although the cache structure shown in Figure C.1 provides good performance for repeated reads and writes from a given CPU to a given item of data, its performance for the first write to a given cache line is quite poor. To see this, consider Figure C.4, which shows a timeline of a write by CPU 0 to a cacheline held in CPU 1's cache. Since CPU 0 must wait for the cache line to arrive before it can write to it, CPU 0 must stall for an extended period of time.[3]

But there is no real reason to force CPU 0 to stall for so long—after all, regardless of what data happens to be in the cache line that CPU 1 sends it, CPU 0 is going to unconditionally overwrite it.

[3] The time required to transfer a cache line from one CPU's cache to another's is typically a few orders of magnitude more than that required to execute a simple register-to-register instruction.

www.dbooks.org

Table C.1: Cache Coherence Example

			CPU Cache				Memory	
Sequence #	CPU #	Operation	0	1	2	3	0	8
0		Initial State	–/I	–/I	–/I	–/I	V	V
1	0	Load	0/S	–/I	–/I	–/I	V	V
2	3	Load	0/S	–/I	–/I	0/S	V	V
3	0	Invalidation	8/S	–/I	–/I	0/S	V	V
4	2	RMW	8/S	–/I	0/E	–/I	V	V
5	2	Store	8/S	–/I	0/M	–/I	I	V
6	1	Atomic Inc	8/S	0/M	–/I	–/I	I	V
7	1	Writeback	8/S	8/S	–/I	–/I	V	V

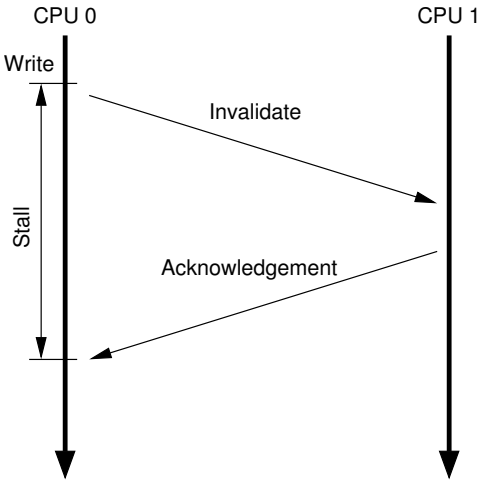


Figure C.4: Writes See Unnecessary Stalls

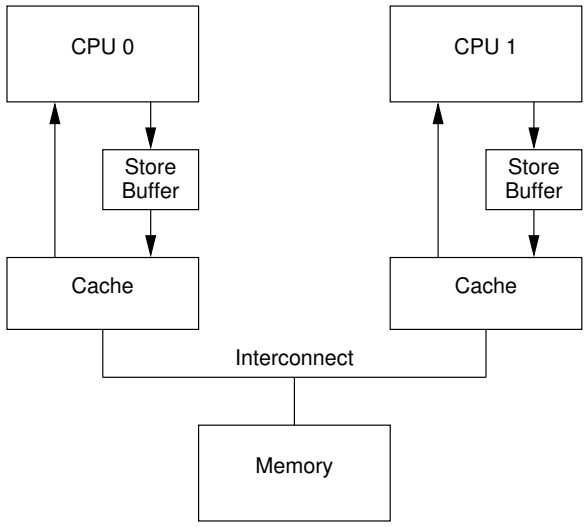


Figure C.5: Caches With Store Buffers

C.3.1 Store Buffers

One way to prevent this unnecessary stalling of writes is to add "store buffers" between each CPU and its cache, as shown in Figure C.5. With the addition of these store buffers, CPU 0 can simply record its write in its store buffer and continue executing. When the cache line does finally make its way from CPU 1 to CPU 0, the data will be moved from the store buffer to the cache line.

Quick Quiz C.7: But if the main purpose of store buffers is to hide acknowledgment latencies in multiprocessor cache-coherence protocols, why do uniprocessors also have store buffers? ■

These store buffers are local to a given CPU or, on systems with hardware multithreading, local to a given core. Either way, a given CPU is permitted to access only the store buffer assigned to it. For example, in Figure C.5, CPU 0 cannot access CPU 1's store buffer and vice versa. This restriction simplifies the hardware by separating concerns: The store buffer improves performance for consecutive writes, while the responsibility for communicating among CPUs (or cores, as the case may be) is fully shouldered by the cache-coherence protocol. However, even given this restriction, there are complications that must be addressed, which are covered in the next two sections.

C.3.2 Store Forwarding

To see the first complication, a violation of self-consistency, consider the following code with variables

"a" and "b" both initially zero, and with the cache line containing variable "a" initially owned by CPU 1 and that containing "b" initially owned by CPU 0:

```
1 a = 1;
2 b = a + 1;
3 assert(b == 2);
```

One would not expect the assertion to fail. However, if one were foolish enough to use the very simple architecture shown in Figure C.5, one would be surprised. Such a system could potentially see the following sequence of events:

1 CPU 0 starts executing the `a = 1`.

2 CPU 0 looks "a" up in the cache, and finds that it is missing.

3 CPU 0 therefore sends a "read invalidate" message in order to get exclusive ownership of the cache line containing "a".

4 CPU 0 records the store to "a" in its store buffer.

5 CPU 1 receives the "read invalidate" message, and responds by transmitting the cache line and removing that cacheline from its cache.

6 CPU 0 starts executing the `b = a + 1`.

7 CPU 0 receives the cache line from CPU 1, which still has a value of zero for "a".

8 CPU 0 loads "a" from its cache, finding the value zero.

9 CPU 0 applies the entry from its store buffer to the newly arrived cache line, setting the value of "a" in its cache to one.

10 CPU 0 adds one to the value zero loaded for "a" above, and stores it into the cache line containing "b" (which we will assume is already owned by CPU 0).

11 CPU 0 executes `assert(b == 2)`, which fails.

The problem is that we have two copies of "a", one in the cache and the other in the store buffer.

This example breaks a very important guarantee, namely that each CPU will always see its own operations as if they happened in program order. Breaking this guarantee is violently counter-intuitive to software types, so much so that the hardware guys took pity and implemented "store forwarding", where each CPU refers to (or "snoops") its store buffer as well as its cache when performing loads, as shown in Figure C.6. In other words, a given CPU's stores are directly forwarded to its subsequent loads, without having to pass through the cache.

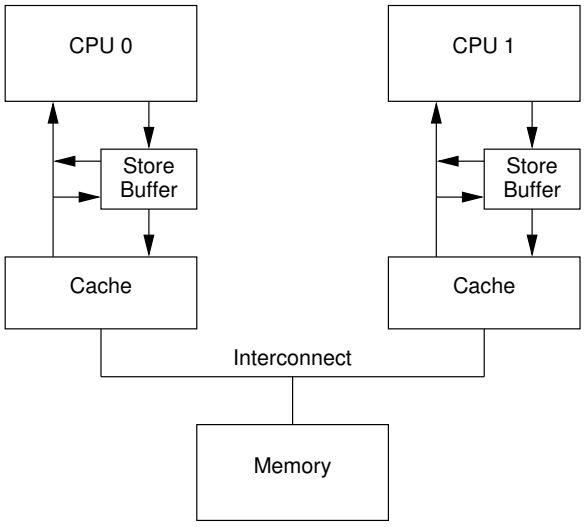


Figure C.6: Caches With Store Forwarding

With store forwarding in place, item 8 in the above sequence would have found the correct value of 1 for "a" in the store buffer, so that the final value of "b" would have been 2, as one would hope.

C.3.3 Store Buffers and Memory Barriers

To see the second complication, a violation of global memory ordering, consider the following code sequences with variables "a" and "b" initially zero:

```
1  void foo(void)
2  {
3    a = 1;
4    b = 1;
5  }
6
7  void bar(void)
8  {
9    while (b == 0) continue;
10   assert(a == 1);
11 }
```

Suppose CPU 0 executes foo() and CPU 1 executes bar(). Suppose further that the cache line containing "a" resides only in CPU 1's cache, and that the cache line containing "b" is owned by CPU 0. Then the sequence of operations might be as follows:

www.dbooks.org

1 CPU 0 executes `a = 1`. The cache line is not in CPU 0's cache, so CPU 0 places the new value of "a" in its store buffer and transmits a "read invalidate" message.

2 CPU 1 executes `while (b == 0)continue`, but the cache line containing "b" is not in its cache. It therefore transmits a "read" message.

3 CPU 0 executes `b = 1`. It already owns this cache line (in other words, the cache line is already in either the "modified" or the "exclusive" state), so it stores the new value of "b" in its cache line.

4 CPU 0 receives the "read" message, and transmits the cache line containing the now-updated value of "b" to CPU 1, also marking the line as "shared" in its own cache.

5 CPU 1 receives the cache line containing "b" and installs it in its cache.

6 CPU 1 can now finish executing `while (b == 0)continue`, and since it finds that the value of "b" is 1, it proceeds to the next statement.

7 CPU 1 executes the `assert(a == 1)`, and, since CPU 1 is working with the old value of "a", this assertion fails.

8 CPU 1 receives the "read invalidate" message, and transmits the cache line containing "a" to CPU 0 and invalidates this cache line from its own cache. But it is too late.

9 CPU 0 receives the cache line containing "a" and applies the buffered store just in time to fall victim to CPU 1's failed assertion.

Quick Quiz C.8: In step 1 above, why does CPU 0 need to issue a "read invalidate" rather than a simple "invalidate"? ■

The hardware designers cannot help directly here, since the CPUs have no idea which variables are related, let alone how they might be related. Therefore, the hardware designers provide memory-barrier instructions to allow the software to tell the CPU about such relations. The program fragment must be updated to contain the memory barrier:

```
1  void foo(void)
2  {
3    a = 1;
4    smp_mb();
5    b = 1;
6  }
7
8  void bar(void)
9  {
10   while (b == 0) continue;
11   assert(a == 1);
12 }
```

The memory barrier `smp_mb()` will cause the CPU to flush its store buffer before applying each subsequent store to its variable's cache line. The CPU could either simply stall until the store buffer was empty before proceeding, or it could use the store buffer to hold subsequent stores until all of the prior entries in the store buffer had been applied.

With this latter approach the sequence of operations might be as follows:

1 CPU 0 executes `a = 1`. The cache line is not in CPU 0's cache, so CPU 0 places the new value of "a" in its store buffer and transmits a "read invalidate" message.

2 CPU 1 executes `while (b == 0)continue`, but the cache line containing "b" is not in its cache. It therefore transmits a "read" message.

3 CPU 0 executes `smp_mb()`, and marks all current store-buffer entries (namely, the `a = 1`).

4 CPU 0 executes `b = 1`. It already owns this cache line (in other words, the cache line is already in either the "modified" or the "exclusive" state), but there is a marked entry in the store buffer. Therefore, rather than store the new value of "b" in the cache line, it instead places it in the store buffer (but in an *unmarked* entry).

5 CPU 0 receives the "read" message, and transmits the cache line containing the original value of "b" to CPU 1. It also marks its own copy of this cache line as "shared".

6 CPU 1 receives the cache line containing "b" and installs it in its cache.

7 CPU 1 can now load the value of "b", but since it finds that the value of "b" is still 0, it repeats the

while statement. The new value of "b" is safely hidden in CPU 0's store buffer.

8 CPU 1 receives the "read invalidate" message, and transmits the cache line containing "a" to CPU 0 and invalidates this cache line from its own cache.

9 CPU 0 receives the cache line containing "a" and applies the buffered store, placing this line into the "modified" state.

10 Since the store to "a" was the only entry in the store buffer that was marked by the smp_mb(), CPU 0 can also store the new value of "b"—except for the fact that the cache line containing "b" is now in "shared" state.

11 CPU 0 therefore sends an "invalidate" message to CPU 1.

12 CPU 1 receives the "invalidate" message, invalidates the cache line containing "b" from its cache, and sends an "acknowledgement" message to CPU 0.

13 CPU 1 executes while (b == 0)continue, but the cache line containing "b" is not in its cache. It therefore transmits a "read" message to CPU 0.

14 CPU 0 receives the "acknowledgement" message, and puts the cache line containing "b" into the "exclusive" state. CPU 0 now stores the new value of "b" into the cache line.

15 CPU 0 receives the "read" message, and transmits the cache line containing the new value of "b" to CPU 1. It also marks its own copy of this cache line as "shared".

16 CPU 1 receives the cache line containing "b" and installs it in its cache.

17 CPU 1 can now load the value of "b", and since it finds that the value of "b" is 1, it exits the while loop and proceeds to the next statement.

18 CPU 1 executes the assert(a == 1), but the cache line containing "a" is no longer in its cache. Once it gets this cache from CPU 0, it will be working with the up-to-date value of "a", and the assertion therefore passes.

Quick Quiz C.9: After step 15 in Appendix C.3.3 on page 415, both CPUs might drop the cache line containing the new value of "b". Wouldn't that cause this new value to be lost? ■

As you can see, this process involves no small amount of bookkeeping. Even something intuitively simple, like "load the value of a" can involve lots of complex steps in silicon.

C.4 Store Sequences Result in Unnecessary Stalls

Unfortunately, each store buffer must be relatively small, which means that a CPU executing a modest sequence of stores can fill its store buffer (for example, if all of them result in cache misses). At that point, the CPU must once again wait for invalidations to complete in order to drain its store buffer before it can continue executing. This same situation can arise immediately after a memory barrier, when *all* subsequent store instructions must wait for invalidations to complete, regardless of whether or not these stores result in cache misses.

This situation can be improved by making invalidate acknowledge messages arrive more quickly. One way of accomplishing this is to use per-CPU queues of invalidate messages, or "invalidate queues".

C.4.1 Invalidate Queues

One reason that invalidate acknowledge messages can take so long is that they must ensure that the corresponding cache line is actually invalidated, and this invalidation can be delayed if the cache is busy, for example, if the CPU is intensively loading and storing data, all of which resides in the cache. In addition, if a large number of invalidate messages arrive in a short time period, a given CPU might fall behind in processing them, thus possibly stalling all the other CPUs.

However, the CPU need not actually invalidate the cache line before sending the acknowledgement. It could instead queue the invalidate message with the understanding that the message will be processed before the CPU sends any further messages regarding that cache line.

C.4.2 Invalidate Queues and Invalidate Acknowledge

Figure C.7 shows a system with invalidate queues. A CPU with an invalidate queue may acknowledge an invalidate message as soon as it is placed in the queue, instead

www.dbooks.org

of having to wait until the corresponding line is actually invalidated. Of course, the CPU must refer to its invalidate queue when preparing to transmit invalidation messages—if an entry for the corresponding cache line is in the invalidate queue, the CPU cannot immediately transmit the invalidate message; it must instead wait until the invalidate-queue entry has been processed.

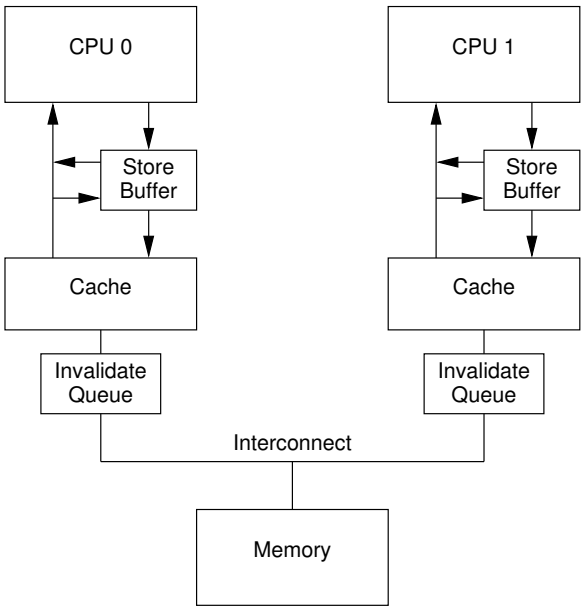


Figure C.7: Caches With Invalidate Queues

Placing an entry into the invalidate queue is essentially a promise by the CPU to process that entry before transmitting any MESI protocol messages regarding that cache line. As long as the corresponding data structures are not highly contended, the CPU will rarely be inconvenienced by such a promise.

However, the fact that invalidate messages can be buffered in the invalidate queue provides additional opportunity for memory-misordering, as discussed in the next section.

C.4.3 Invalidate Queues and Memory Barriers

Let us suppose that CPUs queue invalidation requests, but respond to them immediately. This approach minimizes the cache-invalidation latency seen by CPUs doing stores, but can defeat memory barriers, as seen in the following example.

Suppose the values of "a" and "b" are initially zero, that "a" is replicated read-only (MESI "shared" state), and that "b" is owned by CPU 0 (MESI "exclusive" or "modified" state). Then suppose that CPU 0 executes `foo()` while CPU 1 executes function `bar()` in the following code fragment:

```
1  void foo(void)
2  {
3    a = 1;
4    smp_mb();
5    b = 1;
6  }
7
8  void bar(void)
9  {
10   while (b == 0) continue;
11   assert(a == 1);
12 }
```

Then the sequence of operations might be as follows:

1 CPU 0 executes `a = 1`. The corresponding cache line is read-only in CPU 0's cache, so CPU 0 places the new value of "a" in its store buffer and transmits an "invalidate" message in order to flush the corresponding cache line from CPU 1's cache.

2 CPU 1 executes `while (b == 0)continue`, but the cache line containing "b" is not in its cache. It therefore transmits a "read" message.

3 CPU 1 receives CPU 0's "invalidate" message, queues it, and immediately responds to it.

4 CPU 0 receives the response from CPU 1, and is therefore free to proceed past the `smp_mb()` on line 4 above, moving the value of "a" from its store buffer to its cache line.

5 CPU 0 executes `b = 1`. It already owns this cache line (in other words, the cache line is already in either the "modified" or the "exclusive" state), so it stores the new value of "b" in its cache line.

6 CPU 0 receives the "read" message, and transmits the cache line containing the now-updated value of "b" to CPU 1, also marking the line as "shared" in its own cache.

7 CPU 1 receives the cache line containing "b" and installs it in its cache.

8 CPU 1 can now finish executing `while (b == 0)continue`, and since it finds that the value of "b" is 1, it proceeds to the next statement.

9 CPU 1 executes the `assert(a == 1)`, and, since the old value of "a" is still in CPU 1's cache, this assertion fails.

10 Despite the assertion failure, CPU 1 processes the queued "invalidate" message, and (tardily) invalidates the cache line containing "a" from its own cache.

Quick Quiz C.10: In step 1 of the first scenario in Appendix C.4.3, why is an "invalidate" sent instead of a "read invalidate" message? Doesn't CPU 0 need the values of the other variables that share this cache line with "a"? ■

There is clearly not much point in accelerating invalidation responses if doing so causes memory barriers to effectively be ignored. However, the memory-barrier instructions can interact with the invalidate queue, so that when a given CPU executes a memory barrier, it marks all the entries currently in its invalidate queue, and forces any subsequent load to wait until all marked entries have been applied to the CPU's cache. Therefore, we can add a memory barrier to function `bar` as follows:

```
1  void foo(void)
2  {
3    a = 1;
4    smp_mb();
5    b = 1;
6  }
7
8  void bar(void)
9  {
10   while (b == 0) continue;
11   smp_mb();
12   assert(a == 1);
13 }
```

Quick Quiz C.11: Say what??? Why do we need a memory barrier here, given that the CPU cannot possibly execute the `assert()` until after the `while` loop completes? ■

With this change, the sequence of operations might be as follows:

1 CPU 0 executes `a = 1`. The corresponding cache line is read-only in CPU 0's cache, so CPU 0 places the new value of "a" in its store buffer and transmits an "invalidate" message in order to flush the corresponding cache line from CPU 1's cache.

2 CPU 1 executes `while (b == 0)continue`, but the cache line containing "b" is not in its cache. It therefore transmits a "read" message.

3 CPU 1 receives CPU 0's "invalidate" message, queues it, and immediately responds to it.

4 CPU 0 receives the response from CPU 1, and is therefore free to proceed past the `smp_mb()` on line 4 above, moving the value of "a" from its store buffer to its cache line.

5 CPU 0 executes `b = 1`. It already owns this cache line (in other words, the cache line is already in either the "modified" or the "exclusive" state), so it stores the new value of "b" in its cache line.

6 CPU 0 receives the "read" message, and transmits the cache line containing the now-updated value of "b" to CPU 1, also marking the line as "shared" in its own cache.

7 CPU 1 receives the cache line containing "b" and installs it in its cache.

8 CPU 1 can now finish executing `while (b == 0)continue`, and since it finds that the value of "b" is 1, it proceeds to the next statement, which is now a memory barrier.

9 CPU 1 must now stall until it processes all pre-existing messages in its invalidation queue.

10 CPU 1 now processes the queued "invalidate" message, and invalidates the cache line containing "a" from its own cache.

11 CPU 1 executes the `assert(a == 1)`, and, since the cache line containing "a" is no longer in CPU 1's cache, it transmits a "read" message.

12 CPU 0 responds to this "read" message with the cache line containing the new value of "a".

13 CPU 1 receives this cache line, which contains a value of 1 for "a", so that the assertion does not trigger.

With much passing of MESI messages, the CPUs arrive at the correct answer. This section illustrates why CPU designers must be extremely careful with their cache-coherence optimizations.

C.5 Read and Write Memory Barriers

In the previous section, memory barriers were used to mark entries in both the store buffer and the invalidate

www.dbooks.org

queue. But in our code fragment, `foo()` had no reason to do anything with the invalidate queue, and `bar()` similarly had no reason to do anything with the store buffer.

Many CPU architectures therefore provide weaker memory-barrier instructions that do only one or the other of these two. Roughly speaking, a "read memory barrier" marks only the invalidate queue and a "write memory barrier" marks only the store buffer, while a full-fledged memory barrier does both.

The effect of this is that a read memory barrier orders only loads on the CPU that executes it, so that all loads preceding the read memory barrier will appear to have completed before any load following the read memory barrier. Similarly, a write memory barrier orders only stores, again on the CPU that executes it, and again so that all stores preceding the write memory barrier will appear to have completed before any store following the write memory barrier. A full-fledged memory barrier orders both loads and stores, but again only on the CPU executing the memory barrier.

If we update `foo` and `bar` to use read and write memory barriers, they appear as follows:

```
1  void foo(void)
2  {
3    a = 1;
4    smp_wmb();
5    b = 1;
6  }
7
8  void bar(void)
9  {
10   while (b == 0) continue;
11   smp_rmb();
12   assert(a == 1);
13 }
```

Some computers have even more flavors of memory barriers, but understanding these three variants will provide a good introduction to memory barriers in general.

C.6 Example Memory-Barrier Sequences

This section presents some seductive but subtly broken uses of memory barriers. Although many of them will work most of the time, and some will work all the time on some specific CPUs, these uses must be avoided if the goal is to produce code that works reliably on all CPUs. To help us better see the subtle breakage, we first need to focus on an ordering-hostile architecture.

C.6.1 Ordering-Hostile Architecture

A number of ordering-hostile computer systems have been produced over the decades, but the nature of the hostility has always been extremely subtle, and understanding it has required detailed knowledge of the specific hardware. Rather than picking on a specific hardware vendor, and as a presumably attractive alternative to dragging the reader through detailed technical specifications, let us instead design a mythical but maximally memory-ordering-hostile computer architecture.[4]

This hardware must obey the following ordering constraints [McK05a, McK05b]:

1. Each CPU will always perceive its own memory accesses as occurring in program order.
2. CPUs will reorder a given operation with a store only if the two operations are referencing different locations.
3. All of a given CPU's loads preceding a read memory barrier (`smp_rmb()`) will be perceived by all CPUs to precede any loads following that read memory barrier.
4. All of a given CPU's stores preceding a write memory barrier (`smp_wmb()`) will be perceived by all CPUs to precede any stores following that write memory barrier.
5. All of a given CPU's accesses (loads and stores) preceding a full memory barrier (`smp_mb()`) will be perceived by all CPUs to precede any accesses following that memory barrier.

Quick Quiz C.12: Does the guarantee that each CPU sees its own memory accesses in order also guarantee that each user-level thread will see its own memory accesses in order? Why or why not? ■

Imagine a large non-uniform cache architecture (NUCA) system that, in order to provide fair allocation of interconnect bandwidth to CPUs in a given node, provided per-CPU queues in each node's interconnect interface, as shown in Figure C.8. Although a given CPU's accesses are ordered as specified by memory barriers executed by that CPU, however, the relative order of a given pair of

[4] Readers preferring a detailed look at real hardware architectures are encouraged to consult CPU vendors' manuals [SW95, Adv02, Int02b, IBM94, LHF05, SPA94, Int04b, Int04a, Int04c], Gharachorloo's dissertation [Gha95], Peter Sewell's work [Sew], or the excellent hardware-oriented primer by Sorin, Hill, and Wood [SHW11].

CPUs' accesses could be severely reordered, as we will see.[5]

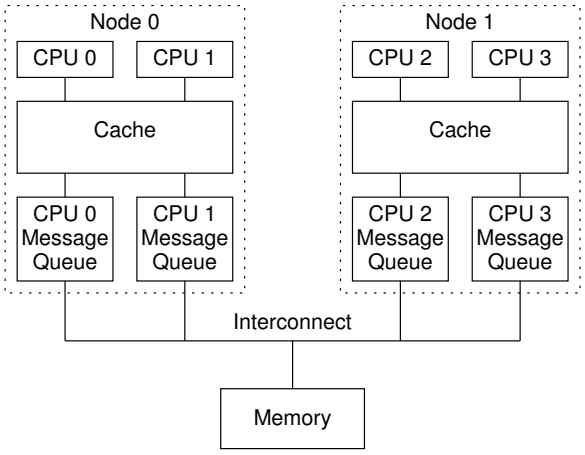


Figure C.8: Example Ordering-Hostile Architecture

C.6.2 Example 1

Listing C.1 shows three code fragments, executed concurrently by CPUs 0, 1, and 2. Each of "a", "b", and "c" are initially zero.

Listing C.1: Memory Barrier Example 1

CPU 0	CPU 1	CPU 2
`a = 1;`	`while (b == 0);`	
`smp_wmb();`	`c = 1;`	`z = c;`
`b = 1;`		`smp_rmb();`
		`x = a;`
		`assert(z == 0 || x == 1);`

Suppose CPU 0 recently experienced many cache misses, so that its message queue is full, but that CPU 1 has been running exclusively within the cache, so that its message queue is empty. Then CPU 0's assignment to "a" and "b" will appear in Node 0's cache immediately (and thus be visible to CPU 1), but will be blocked behind CPU 0's prior traffic. In contrast, CPU 1's assignment to "c" will sail through CPU 1's previously empty queue. Therefore, CPU 2 might well see CPU 1's assignment to "c" before it sees CPU 0's assignment to "a", causing the assertion to fire, despite the memory barriers.

[5] Any real hardware architect or designer will no doubt be objecting strenuously, as they just might be a bit upset about the prospect of working out which queue should handle a message involving a cache line that both CPUs accessed, to say nothing of the many races that this example poses. All I can say is "Give me a better example".

Therefore, portable code cannot rely on this assertion not firing, as both the compiler and the CPU can reorder the code so as to trip the assertion.

Quick Quiz C.13: Could this code be fixed by inserting a memory barrier between CPU 1's "while" and assignment to "c"? Why or why not? ■

C.6.3 Example 2

Listing C.2 shows three code fragments, executed concurrently by CPUs 0, 1, and 2. Both "a" and "b" are initially zero.

Listing C.2: Memory Barrier Example 2

CPU 0	CPU 1	CPU 2
`a = 1;`	`while (a == 0);`	
	`smp_mb();`	`y = b;`
	`b = 1;`	`smp_rmb();`
		`x = a;`
		`assert(y == 0 || x == 1);`

Again, suppose CPU 0 recently experienced many cache misses, so that its message queue is full, but that CPU 1 has been running exclusively within the cache, so that its message queue is empty. Then CPU 0's assignment to "a" will appear in Node 0's cache immediately (and thus be visible to CPU 1), but will be blocked behind CPU 0's prior traffic. In contrast, CPU 1's assignment to "b" will sail through CPU 1's previously empty queue. Therefore, CPU 2 might well see CPU 1's assignment to "b" before it sees CPU 0's assignment to "a", causing the assertion to fire, despite the memory barriers.

In theory, portable code should not rely on this example code fragment, however, as before, in practice it actually does work on most mainstream computer systems.

C.6.4 Example 3

Listing C.3 shows three code fragments, executed concurrently by CPUs 0, 1, and 2. All variables are initially zero.

Note that neither CPU 1 nor CPU 2 can proceed to line 5 until they see CPU 0's assignment to "b" on line 3. Once CPU 1 and 2 have executed their memory barriers on line 4, they are both guaranteed to see all assignments by CPU 0 preceding its memory barrier on line 2. Similarly, CPU 0's memory barrier on line 8 pairs with those of CPUs 1 and 2 on line 4, so that CPU 0 will not execute the assignment to "e" on line 9 until after its assignment

www.dbooks.org

Listing C.3: Memory Barrier Example 3

	CPU 0	CPU 1	CPU 2
1	`a = 1;`		
2	`smp_wmb();`		
3	`b = 1;`	`while (b == 0);`	`while (b == 0);`
4		`smp_mb();`	`smp_mb();`
5		`c = 1;`	`d = 1;`
6	`while (c == 0);`		
7	`while (d == 0);`		
8	`smp_mb();`		
9	`e = 1;`		`assert(e == 0 || a == 1);`

to "b" is visible to both of the other CPUs. Therefore, CPU 2's assertion on line 9 is guaranteed *not* to fire.

Quick Quiz C.14: Suppose that lines 3–5 for CPUs 1 and 2 in Listing C.3 are in an interrupt handler, and that the CPU 2's line 9 runs at process level. In other words, the code in all three columns of the table runs on the same CPU, but the first two columns run in an interrupt handler, and the third column runs at process level, so that the code in third column can be interrupted by the code in the first two columns. What changes, if any, are required to enable the code to work correctly, in other words, to prevent the assertion from firing? ■

Quick Quiz C.15: If CPU 2 executed an `assert(e==0||c==1)` in the example in Listing C.3, would this assert ever trigger? ■

The Linux kernel's `synchronize_rcu()` primitive uses an algorithm similar to that shown in this example.

C.7 Are Memory Barriers Forever?

There have been a number of recent systems that are significantly less aggressive about out-of-order execution in general and re-ordering memory references in particular. Will this trend continue to the point where memory barriers are a thing of the past?

The argument in favor would cite proposed massively multi-threaded hardware architectures, so that each thread would wait until memory was ready, with tens, hundreds, or even thousands of other threads making progress in the meantime. In such an architecture, there would be no need for memory barriers, because a given thread would simply wait for all outstanding operations to complete before proceeding to the next instruction. Because there would be potentially thousands of other threads, the CPU would be completely utilized, so no CPU time would be wasted.

The argument against would cite the extremely limited number of applications capable of scaling up to a thousand threads, as well as increasingly severe realtime requirements, which are in the tens of microseconds for some applications. The realtime-response requirements are difficult enough to meet as is, and would be even more difficult to meet given the extremely low single-threaded throughput implied by the massive multi-threaded scenarios.

Another argument in favor would cite increasingly sophisticated latency-hiding hardware implementation techniques that might well allow the CPU to provide the illusion of fully sequentially consistent execution while still providing almost all of the performance advantages of out-of-order execution. A counter-argument would cite the increasingly severe power-efficiency requirements presented both by battery-operated devices and by environmental responsibility.

Who is right? We have no clue, so we are preparing to live with either scenario.

C.8 Advice to Hardware Designers

There are any number of things that hardware designers can do to make the lives of software people difficult. Here is a list of a few such things that we have encountered in the past, presented here in the hope that it might help prevent future such problems:

1. I/O devices that ignore cache coherence.

   This charming misfeature can result in DMAs from memory missing recent changes to the output buffer, or, just as bad, cause input buffers to be overwritten by the contents of CPU caches just after the DMA completes. To make your system work in face of such misbehavior, you must carefully flush the CPU caches of any location in any DMA buffer before presenting that buffer to the I/O device. Similarly, you need to flush the CPU caches of any location in any DMA buffer after DMA to that buffer completes. And even then, you need to be *very* careful to avoid

pointer bugs, as even a misplaced read to an input buffer can result in corrupting the data input!

2. External busses that fail to transmit cache-coherence data.

   This is an even more painful variant of the above problem, but causes groups of devices—and even memory itself—to fail to respect cache coherence. It is my painful duty to inform you that as embedded systems move to multicore architectures, we will no doubt see a fair number of such problems arise. Hopefully these problems will clear up by the year 2015.

3. Device interrupts that ignore cache coherence.

   This might sound innocent enough—after all, interrupts aren't memory references, are they? But imagine a CPU with a split cache, one bank of which is extremely busy, therefore holding onto the last cacheline of the input buffer. If the corresponding I/O-complete interrupt reaches this CPU, then that CPU's memory reference to the last cache line of the buffer could return old data, again resulting in data corruption, but in a form that will be invisible in a later crash dump. By the time the system gets around to dumping the offending input buffer, the DMA will most likely have completed.

4. Inter-processor interrupts (IPIs) that ignore cache coherence.

   This can be problematic if the IPI reaches its destination before all of the cache lines in the corresponding message buffer have been committed to memory.

5. Context switches that get ahead of cache coherence.

   If memory accesses can complete too wildly out of order, then context switches can be quite harrowing. If the task flits from one CPU to another before all the memory accesses visible to the source CPU make it to the destination CPU, then the task could easily see the corresponding variables revert to prior values, which can fatally confuse most algorithms.

6. Overly kind simulators and emulators.

   It is difficult to write simulators or emulators that force memory re-ordering, so software that runs just fine in these environments can get a nasty surprise when it first runs on the real hardware. Unfortunately, it is still the rule that the hardware is more devious than are the simulators and emulators, but we hope that this situation changes.

Again, we encourage hardware designers to avoid these practices!

www.dbooks.org

De gustibus non est disputandum.

Latin maxim

Appendix D

Style Guide

This appendix is a collection of style guides which is intended as a reference to improve consistency in perfbook. It also contains several suggestions and their experimental examples.

Appendix D.1 describes basic punctuation and spelling rules. Appendix D.2 explains rules related to unit symbols. Appendix D.3 summarizes LaTeX-specific conventions.

D.1 Paul's Conventions

Following is the list of Paul's conventions assembled from his answers to Akira's questions regarding perfbook's punctuation policy.

- (On punctuations and quotations) Despite being American myself, for this sort of book, the UK approach is better because it removes ambiguities like the following:

  > Type "`ls -a`," look for the file "`.`," and file a bug if you don't see it.

  The following is much more clear:

  > Type "`ls -a`", look for the file "`.`", and file a bug if you don't see it.

- American English spelling: "color" rather than "colour".

- Oxford comma: "a, b, and c" rather than "a, b and c". This is arbitrary. Cases where the Oxford comma results in ambiguity should be reworded, for example, by introducing numbering: "a, b, and c and d" should be "(1) a, (2) b, and (3) c and d".

- Italic for emphasis. Use sparingly.

- `\co{}` for identifiers, `\url{}` for URLs, `\path{}` for filenames.

- Dates should use an unambiguous format. Never "mm/dd/yy" or "dd/mm/yy", but rather "July 26, 2016" or "26 July 2016" or "26-Jul-2016" or "2016/07/26". I tend to use `yyyy.mm.ddA` for filenames, for example.

- North American rules on periods and abbreviations. For example neither of the following can reasonably be interpreted as two sentences:

  – Say hello, to Mr. Jones.

  – If it looks like she sprained her ankle, call Dr. Smith and then tell her to keep the ankle iced and elevated.

  An ambiguous example:

  > If I take the cow, the pig, the horse, etc. George will be upset.

  can be written with more words:

  > If I take the cow, the pig, the horse, or much of anything else, George will be upset.

  or:

  > If I take the cow, the pig, the horse, etc., George will be upset.

- I don't like ampersand ("&") in headings, but will sometimes use it if doing so prevents a line break in that heading.

- When mentioning words, I use quotations. When introducing a new word, I use `\emph{}`.

Edition.2-rc11

www.dbooks.org

Following is a convention regarding punctuation in LaTeX sources.

- Place a newline after a colon (`:`) and the end of a sentence. This avoids the whole one-space/two-space food fight and also has the advantage of more clearly showing changes to single sentences in the middle of long paragraphs.

D.2 NIST Style Guide

D.2.1 Unit Symbol

D.2.1.1 SI Unit Symbol

NIST style guide[1] states the following rules (rephrased for perfbook).

- When SI unit symbols such as "ns", "MHz", and "K" (kelvin) are used behind numerical values, narrow spaces should be placed between the values and the symbols.

  A narrow space can be coded in LaTeX by the sequence of "`\,`". For example,

  > "2.4 GHz", rather then "2.4GHz".

- Even when the value is used in adjectival sense, a narrow space should be placed. For example,

  > "a 10 ms interval", rather than "a 10-ms interval" nor "a 10ms interval".

The symbol of micro (μ:10^{-6}) can be typeset easily by the help of "gensymb" LaTeX package. A macro "`\micro`" can be used in both text and math modes. To typeset the symbol of "microsecond", you can do so by "`\micro s`". For example,

> 10 µs

Note that math mode "`\mu`" is italic by default and should not be used as a prefix. An improper example:

> 10 μs (math mode "`\mu`")

[1] https://www.nist.gov/pml/nist-guide-si-chapter-7-rules-and-style-conventions-expressing-values-quantities

D.2.1.2 Non-SI Unit Symbol

Although NIST style guide does not cover non-SI unit symbols such as "KB", "MB", and "GB", the same rule should be followed.

Example:

> "A 240 GB hard drive", rather than "a 240-GB hard drive" nor "a 240GB hard drive".

Strictly speaking, NIST guide requires us to use the binary prefixes "Ki", "Mi", or "Gi" to represent powers of 2^{10}. However, we accept the JEDEC conventions to use "K", "M", and "G" as binary prefixes in describing memory capacity.[2]

An acceptable example:

> "8 GB of main memory", meaning "8 GiB of main memory".

Also, it is acceptable to use just "K", "M", or "G" as abbreviations appended to a numerical value, e.g., "4K entries". In such cases, no space before an abbreviation is required. For example,

> "8K entries", rather than "8 K entries".

If you put a space in between, the symbol looks like a unit symbol and is confusing. Note that "K" and "k" represent 2^{10} and 10^3, respectively. "M" can represent either 2^{20} or 10^6, and "G" can represent either 2^{30} or 10^9. These ambiguities should not be confusing in discussing approximate order.

D.2.1.3 Degree Symbol

The angular-degree symbol (°) does not require any space in front of it. NIST style guide clearly states so.

The symbol of degree can also be typeset easily by the help of gensymb package. A macro "`\degree`" can be used in both text and math modes.

Example:

> 45°, rather than 45 °.

D.2.1.4 Percent Symbol

NIST style guide treats the percent symbol (%) as the same as SI unit symbols.

> 50 % possibility, rather than 50% possibility.

[2] https://www.jedec.org/standards-documents/dictionary/terms/mega-m-prefix-units-semiconductor-storage-capacity

D.2.1.5 Font Style

Quote from NIST check list:[3]

> Variables and quantity symbols are in italic type. Unit symbols are in roman type. Numbers should generally be written in roman type. These rules apply irrespective of the typeface used in the surrounding text.

For example,

e (elementary charge)

On the other hand, mathematical constants such as the base of natural logarithms should be roman.[4] For example,

e^x

D.2.2 NIST Guide Yet To Be Followed

There are a few cases where NIST style guide is not followed. Other English conventions are followed in such cases.

D.2.2.1 Digit Grouping

Quote from NIST checklist:[5]

> The digits of numerical values having more than four digits on either side of the decimal marker are separated into groups of three using a thin, fixed space counting from both the left and right of the decimal marker. Commas are not used to separate digits into groups of three.

NIST Example:	15 739.012 53 ms
Our convention:	15,739.01253 ms

In LaTeX coding, it is cumbersome to place thin spaces as are recommended in NIST guide. The `\num{}` command provided by the "siunitx" package would be of help for us to follow this rule. It would also help us overcome different conventions. We can select a specific digit-grouping style as a default in preamble, or specify an option to each `\num{}` command as is shown in Table D.1.

Table D.1: Digit-Grouping Style

Style	Outputs of `\num{}`		
NIST/SI (English)	12 345	12.345	1 234 567.89
SI (French)	12 345	12,345	1 234 567,89
English	12,345	12.345	1,234,567.89
French	12 345	12,345	1 234 567,89
Other Europe	12.345	12,345	1.234.567,89

As are evident in Table D.1, periods and commas used as other than decimal markers are confusing and should be avoided, especially in documents expecting global audiences.

By marking up constant decimal values by `\num{}` commands, the LaTeX source would be exempted from any particular conventions.

Because of its open-source policy, this approach should give more "portability" to perfbook.

D.3 LaTeX Conventions

Good looking LaTeX documents require further considerations on proper use of font styles, line break exceptions, etc. This section summarizes guidelines specific to LaTeX.

D.3.1 Monospace Font

Monospace font (or typewriter font) is heavily used in this textbook. First policy regarding monospace font in perfbook is to avoid directly using "`\texttt`" or "`\tt`" macro. It is highly recommended to use a macro or an environment indicating the reason why you want the font.

This section explains the use cases of such macros and environments.

D.3.1.1 Code Snippet

Because the "`verbatim`" environment is a primitive way to include listings, we have transitioned to a scheme which uses the "`fancyvrb`" package for code snippets.

The goal of the scheme is to extract LaTeX sources of code snippets directly from code samples under `CodeSamples` directory. It also makes it possible to embed line labels in the code samples, which can be referenced within the LaTeX sources. This reduces the burden of keeping line numbers in the text consistent with those in code snippets.

Code-snippet extraction is handled by a couple of perl scripts and recipes in Makefile. We use the escaping

[3] #6 in `https://physics.nist.gov/cuu/Units/checklist.html`

[4] `https://physics.nist.gov/cuu/pdf/typefaces.pdf`

[5] #16 in `https://physics.nist.gov/cuu/Units/checklist.html`.

www.dbooks.org

Listing D.1: LaTeX Source of Sample Code Snippet (Current)

```
1  \begin{listing}[tb]
2  \begin{fcvlabel}[ln:base1]
3  \begin{VerbatimL}[commandchars=\$\[\]]
4  /*
5   * Sample Code Snippet
6   */
7  #include <stdio.h>
8  int main(void)
9  {
10         printf("Hello world!\n");       $lnlbl[printf]
11         return 0;                       $lnlbl[return]
12 }
13 \end{VerbatimL}
14 \end{fcvlabel}
15 \caption{Sample Code Snippet}
16 \label{lst:app:styleguide:Sample Code Snippet}
17 \end{listing}
```

Listing D.2: Sample Code Snippet

```
1 /*
2  * Sample Code Snippet
3  */
4 #include <stdio.h>
5 int main(void)
6 {
7   printf("Hello world!\n");
8   return 0;
9 }
```

feature of the `fancyvrb` package to embed line labels as comments.

We used to use the "`verbbox`" environment provided by the "`verbatimbox`" package. Appendix D.3.1.2 describes how `verbbox` can automatically generate line numbers, but those line numbers cannot be referenced within the LaTeX sources.

Let's start by looking at how code snippets are coded in the current scheme. There are three customized environments of "`Verbatim`". "`VerbatimL`" is for floating snippets within the "`listing`" environment. "`VerbatimN`" is for inline snippets with line count enabled. "`VerbatimU`" is for inline snippets without line count. They are defined in the preamble as shown below:

```
\DefineVerbatimEnvironment{VerbatimL}{Verbatim}%
  {fontsize=\scriptsize,numbers=left,numbersep=5pt,%
    xleftmargin=9pt,obeytabs=true,tabsize=2}
\AfterEndEnvironment{VerbatimL}{\vspace*{-9pt}}
\DefineVerbatimEnvironment{VerbatimN}{Verbatim}%
  {fontsize=\scriptsize,numbers=left,numbersep=3pt,%
    xleftmargin=5pt,xrightmargin=5pt,obeytabs=true,%
    tabsize=2,frame=single}
\DefineVerbatimEnvironment{VerbatimU}{Verbatim}%
  {fontsize=\scriptsize,numbers=none,xleftmargin=5pt,%
    xrightmargin=5pt,obeytabs=true,tabsize=2,%
    samepage=true,frame=single}
```

The LaTeX source of a sample code snippet is shown in Listing D.1 and is typeset as shown in Listing D.2.

Labels to lines are specified in "`$lnlbl[]`" command. The characters specified by "`commandchars`" option to `VarbatimL` environment are used by the `fancyvrb` package to substitute "`\lnlbl{}`" for "`$lnlbl[]`". Those characters should be selected so that they don't appear elsewhere in the code snippet.

Labels "`printf`" and "`return`" in Listing D.2 can be referred to as shown below:

```
\begin{fcvref}[ln:base1]
Lines~\lnref{printf} and~\lnref{return} can be referred
to from text.
\end{fcvref}
```

Above code results in the paragraph below:

> Lines 7 and 8 can be referred to from text.

Macros "`\lnlbl{}`" and "`\lnref{}`" are defined in the preamble as follows:

```
\newcommand{\lnlblbase}{}
\newcommand{\lnlbl}[1]{%
  \phantomsection\label{\lnlblbase:#1}}
\newcommand{\lnrefbase}{}
\newcommand{\lnref}[1]{\ref{\lnrefbase:#1}}
```

Environments "`fcvlabel`" and "`fcvref`" are defined as shown below:

```
\newenvironment{fcvlabel}[1][]{%
  \renewcommand{\lnlblbase}{#1}%
  \ignorespaces}{\ignorespacesafterend}
\newenvironment{fcvref}[1][]{%
  \renewcommand{\lnrefbase}{#1}%
  \ignorespaces}{\ignorespacesafterend}
```

The main part of LaTeX source shown on Lines 2–14 in Listing D.1 can be extracted from a code sample of Listing D.3 by a perl script `utilities/fcvextract.pl`. All the relevant rules of extraction are described as recipes in the top level `Makefile` and a script to generate dependencies (`utilities/gen_snippet_d.pl`).

As you can see, Listing D.3 has meta commands in comments of C (C++ style). Those meta commands are interpreted by `utilities/fcvextract.pl`, which distinguishes the type of comment style by the suffix of code sample's file name.

Meta commands which can be used in code samples are listed below:

- `\begin{snippet}[<options>]`
- `\end{snippet}`
- `\lnlbl{<label string>}`
- `\fcvexclude`

Listing D.3: Source of Code Sample with "snippet" Meta Command

```
1 //\begin{snippet}[labelbase=ln:base1,keepcomment=yes,commandchars=\$\[\]]
2 /*
3  * Sample Code Snippet
4  */
5 #include <stdio.h>
6 int main(void)
7 {
8         printf("Hello world!\n");       //\lnlbl{printf}
9         return 0;                       //\lnlbl{return}
10 }
11 //\end{snippet}
```

- `\fcvblank`

"`<options>`" to the `\begin{snippet}` meta command is a comma-spareted list of options shown below:

- `labelbase=<label base string>`
- `keepcomment=yes`
- `gobbleblank=yes`
- `commandchars=\X\Y\Z`

The "`labelbase`" option is mandatory and the string given to it will be passed to the "`\begin{fcvlabel}[<label base string>]`" command as shown on line 2 of Listing D.1. The "`keepcomment=yes`" option tells `fcvextract.pl` to keep comment blocks. Otherwise, comment blocks in C source code will be omitted. The "`gobbleblank=yes`" option will remove empty or blank lines in the resulting snippet. The "`commandchars`" option is given to the `VerbatimL` environment as is. At the moment, it is also mandatory and must come at the end of options listed above. Other types of options, if any, are also passed to the `VerbatimL` environment.

The "`\lnlbl`" commands are converted along the way to reflect the escape-character choice.[6] Source lines with "`\fcvexclude`" are removed. "`\fcvblank`" can be used to keep blank lines when the "`gobbleblank=yes`" option is specified.

There can be multiple pairs of `\begin{snippet}` and `\end{snippet}` as long as they have unique "`labelbase`" strings.

Our naming scheme of "`labelbase`" for unique name space is as follows:

```
ln:<Chapter/Subdirectory>:<File Name>:<Function Name>
```

Litmus tests, which are handled by "`herdtools7`" commands such as "`litmus7`" and "`herd7`", were problematic in this scheme. Those commands have particular rules of where comments can be placed and restriction on permitted characters in comments. They also forbid a couple of tokens to appear in comments. (Tokens in comments might sound strange, but they do have such restriction.)

For example, the first token in a litmus test must be one of "`C`", "`PPC`", "`X86`", "`LISA`", etc., which indicates the flavor of the test. This means no comment is allowed at the beginning of a litmus test.

Similarly, several tokens such as "`exists`", "`filter`", and "`locations`" indicate the end of litmus test's body. Once one of them appears in a litmus test, comments should be of OCaml style ("`(* ... *)`"). Those tokens keep the same meaning even when they appear in comments!

The pair of characters "`{`" and "`}`" also have special meaning in the C flavour tests. They are used to separate portions in a litmus test.

First pair of "`{`" and "`}`" encloses initialization part. Comments in this part should also be in the ocaml form.

You can't use "`{`" and "`}`" in comments in litmus tests, either.

Examples of disallowed comments in a litmus test are shown below:

```
1  // Comment at first
2  C C-sample
3  // Comment with { and } characters
4  {
5  x=2;  // C style comment in initialization
6  }
7
8  P0(int *x}
9  {
10         int r1;
11
12         r1 = READ_ONCE(*x);  // Comment with "exists"
13 }
14
15 [...]
16
17 exists (0:r1=0)  // C++ style comment after test body
```

To avoid parse errors, meta commands in litmus tests (C flavor) are embedded in the following way.

[6] Characters forming comments around the "`\lnlbl`" commands are also gobbled up regardless of the "`keepcomment`" setting.

www.dbooks.org

```
C C-SB+o-o+o-o
//\begin[snippet][labelbase=ln:base,commandchars=\%\@\$]

{
1:r2=0                  (*\lnlbl[initr2]*)
}

P0(int *x0, int *x1)            //\lnlbl[P0:b]
{
        int r2;

        WRITE_ONCE(*x0, 2);
        r2 = READ_ONCE(*x1);
}                               //\lnlbl[P0:e]

P1(int *x0, int *x1)
{
        int r2;

        WRITE_ONCE(*x1, 2);
        r2 = READ_ONCE(*x0);
}

//\end[snippet]
exists (1:r2=0 /\ 0:r2=0)  (* \lnlbl[exists_] *)
```

Example above is converted to the following intermediate code by a script `utilities/reorder_ltms.pl`.[7] The intermediate code can be handled by the common script `utilities/fcvextract.pl`.

```
// Do not edit!
// Generated by utillities/reorder_ltms.pl
//\begin{snippet}[labelbase=ln:base,commandchars=\%\@\$]
C C-SB+o-o+o-o

{
1:r2=0                  //\lnlbl{initr2}
}

P0(int *x0, int *x1)            //\lnlbl{P0:b}
{
        int r2;

        WRITE_ONCE(*x0, 2);
        r2 = READ_ONCE(*x1);
}                               //\lnlbl{P0:e}

P1(int *x0, int *x1)
{
        int r2;

        WRITE_ONCE(*x1, 2);
        r2 = READ_ONCE(*x0);
}

exists (1:r2=0 /\ 0:r2=0)  \lnlbl{exists_}
//\end{snippet}
```

Note that each litmus test's source file can contain at most one pair of `\begin[snippet]` and `\end[snippet]` because of the restriction of comments.

[7] Currently, only C flavor litmus tests are supported.

Listing D.4: LaTeX Source of Sample Code Snippet (Obsolete)

```
\begin{listing}[tb]
{ \scriptsize
\begin{verbbox}[\LstLineNo]
/*
 * Sample Code Snippet
 */
#include <stdio.h>
int main(void)
{
  printf("Hello world!\n");
  return 0;
}
\end{verbbox}
}
\centering
\theverbbox
\caption{Sample Code Snippet (Obsolete)}
\label{lst:app:styleguide:Sample Code Snippet (Obsolete)}
\end{listing}
```

Listing D.5: Sample Code Snippet (Obsolete)

```
1 /*
2  * Sample Code Snippet
3  */
4 #include <stdio.h>
5 int main(void)
6 {
7   printf("Hello world!\n");
8   return 0;
9 }
```

D.3.1.2 Code Snippet (Obsolete)

Sample LaTeX source of a code snippet coded using the "`verbatimbox`" package is shown in Listing D.4 and is typeset as shown in Listing D.5.

The auto-numbering feature of `verbbox` is enabled by the "`\LstLineNo`" macro specified in the option to verbbox (line 3 in Listing D.4). The macro is defined in the preamble of `perfbook.tex` as follows:

```
\newcommand{\LstLineNo}
  {\makebox[5ex][r]{\arabic{VerbboxLineNo}\hspace{2ex}}}
```

The "`verbatim`" environment is used for listings with too many lines to fit in a column. It is also used to avoid overwhelming LaTeX with a lot of floating objects. They are being converted to the scheme using the `VerbatimN` environment.

D.3.1.3 Identifier

We use "`\co{}`" macro for inline identifiers. ("co" stands for "code".)

By putting them into `\co{}`, underscore characters in their names are free of escaping in LaTeX source. It is convenient to search them in source files. Also, `\co{}` macro has a capability to permit line breaks at particular

sequences of letters. Current definition permits a line break at an underscore (_), two consecutive underscores (__), a white space, or an operator ->.

D.3.1.4 Identifier inside Table and Heading

Although \co{} command is convenient for inlining within text, it is fragile because of its capability of line break. When it is used inside a "tabular" environment or its derivative such as "tabularx", it confuses column width estimation of those environments. Furthermore, \co{} can not be safely used in section headings nor description headings.

As a workaround, we use "\tco{}" command inside tables and headings. It has no capability of line break at particular sequences, but still frees us from escaping underscores.

When used in text, \tco{} permits line breaks at white spaces.

D.3.1.5 Other Use Cases of Monospace Font

For URLs, we use "\url{}" command provided by the "hyperref" package. It will generate hyper references to the URLs.

For path names, we use "\path{}" command. It won't generate hyper references.

Both \url{} and \path{} permit line breaks at "/", "-", and ".".[8]

For short monospace statements not to be line broken, we use the "\nbco{}" (non-breakable co) macro.

D.3.1.6 Limitations

There are a few cases where macros introduced in this section do not work as expected. Table D.2 lists such limitations.

Table D.2: Limitation of Monospace Macro

Macro	Need Escape	Should Avoid
\co, \nbco	\, %, {, }	
\tco	#	%, {, }, \

While \co{} requires some characters to be escaped, it can contain any character.

On the other hand, \tco{} can not handle "%", "{", "}", nor "\" properly. If they are escaped by a "\", they appear in the end result with the escape character. The "\verb" command can be used in running text if you need to use monospace font for a string which contains many characters to escape.[9]

D.3.2 Cross-reference

Cross-references to Chapters, Sections, Listings, etc. have been expressed by combinations of names and bare \ref{} commands in the following way:

```
Chapter~\ref{chp:Introduction},
Table~\ref{tab:app:styleguide:Digit-Grouping Style}
```

This is a traditional way of cross-referencing. However, it is tedious and sometimes error-prone to put a name manually on every cross-reference. The cleveref package provides a nicer way of cross-referencing. A few examples follow:

```
\Cref{chp:Introduction},
\cref{sec:intro:Parallel Programming Goals},
\cref{chp:app:styleguide:Style Guide},
\cref{tab:app:styleguide:Digit-Grouping Style}, and
\cref{lst:app:styleguide:Source of Code Sample} are
examples of cross\-/references.
```

Above code is typeset as follows:

> Chapter 2, Section 2.2, Appendix D, Table D.1, and Listing D.3 are examples of cross-references.

As you can see, naming of cross-references is automated. Current setting generates capitalized names for both of \Cref{} and \cref{}, but the former should be used at the beginning of a sentence.

We are in the middle of conversion to cleveref-style cross-referencing.

Cross-references to line numbers of code snippets can be done in a similar way by using \Clnref{} and \clnref{} macros, which mimic cleveref. The former puts "Line" as the name of the reference and the latter "line".

Please refer to cleveref's documentation for further info on its cleverness.

[8] Overfill can be a problem if the URL or the path name contains long runs of unbreakable characters.

[9] The \verb command is not almighty though. For example, you can't use it within a footnote. If you do so, you will see a fatal LaTeX error. A workaround would be a macro named \VerbatimFootnotes provided by the fancyvrb package. Unfortunately, perfbook can't employ it due to the interference with the footnotebackref package.

www.dbooks.org

D.3.3 Non Breakable Spaces

In LaTeX conventions, proper use of non-breakable white spaces is highly recommended. They can prevent widowing and orphaning of single digit numbers or short variable names, which would cause the text to be confusing at first glance.

The thin space mentioned earlier to be placed in front of a unit symbol is non breakable.

Other cases to use a non-breakable space ("~" in LaTeX source, often referred to as "nbsp") are the following (inexhaustive).

- Reference to a Chapter or a Section:

  > Please refer to Appendix D.2.

- Calling out CPU number or Thread name:

  > After they load the pointer, CPUs 1 and 2 will see the stored value.

- Short variable name:

  > The results will be stored in variables `a` and `b`.

D.3.4 Hyphenation and Dashes

D.3.4.1 Hyphenation in Compound Word

In plain LaTeX, compound words such as "high-frequency" can be hyphenated only at the hyphen. This sometimes results in poor typesetting. For example:

> High-frequency radio wave, high-frequency radio wave, high-frequency radio wave, high-frequency radio wave, high-frequency radio wave.

By using a shortcut "`\-/`" provided by the "extdash" package, hyphenation in elements of compound words is enabled in perfbook.[10]

Example with "`\-/`":

> High-frequency radio wave, high-frequency radio wave, high-frequency radio wave, high-frequency radio wave, high-frequency radio wave.

[10] In exchange for enabling the shortcut, we can't use plain LaTeX's shortcut "`\-`" to specify hyphenation points. Use `pfhyphex.tex` to add such exceptions.

D.3.4.2 Non Breakable Hyphen

We want hyphenated compound terms such as "x-coordinate", "y-coordinate", etc. not to be broken at the hyphen following a single letter.

To make a hyphen unbreakable, we can use a short cut "`\=/`" also provided by the "extdash" package.

Example without a shortcut:

> x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates;

Example with "`\-/`":

> x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates;

Example with "`\=/`":

> x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates; x-, y-, and z-coordinates;

Note that "`\=/`" enables hyphenation in elements of compound words as the same as "`\-/`" does.

D.3.4.3 Em Dash

Em dashes are used to indicate parenthetic expression. In perfbook, em dashes are placed without spaces around it. In LaTeX source, an em dash is represented by "`---`".

Example (quote from Appendix C.1):

> This disparity in speed—more than two orders of magnitude—has resulted in the multi-megabyte caches found on modern CPUs.

D.3.4.4 En Dash

In LaTeX convention, en dashes (–) are used for ranges of (mostly) numbers. Past revisions of perfbook didn't follow this rule and used plain dashes (-) for such cases.

Now that `\clnrefrange`, `\crefrange`, and their variants, which generate en dashes, are used for ranges of cross-references, the remaining couple of tens of simple

dashes of other types of ranges have been converted to en dashes for consistency.

Example with a simple dash:

> Lines 4-12 in Listing D.4 are the contents of the verbbox environment. The box is output by the `\theverbbox` macro on line 16.

Example with an en dash:

> Lines 4–12 in Listing D.4 are the contents of the verbbox environment. The box is output by the `\theverbbox` macro on line 16.

D.3.4.5 Numerical Minus Sign

Numerical minus signs should be coded as math mode minus signs, namely "`$-$`".[11] For example,

> −30, rather than -30.

D.3.5 Punctuation

D.3.5.1 Ellipsis

In monospace fonts, ellipses can be expressed by series of periods. For example:

```
Great ... So how do I fix it?
```

However, in proportional fonts, the series of periods is printed with tight spaces as follows:

> Great ... So how do I fix it?

Standard LaTeX defines the `\dots` macro for this purpose. However, it has a kludge in the evenness of spaces. The "ellipsis" package redefines the `\dots` macro to fix the issue.[12] By using `\dots`, the above example is typeset as the following:

> Great . . . So how do I fix it?

Note that the "xspace" option specified to the "ellipsis" package adjusts the spaces after ellipses depending on what follows them.

For example:

- He said, "I . . . really don't remember . . ."
- Sequence A: (one, two, three, . . .)
- Sequence B: (4, 5, . . . , n)

As you can see, extra space is placed before the comma. `\dots` macro can also be used in math mode:

- Sequence C: (1, 2, 3, 5, 8, . . .)
- Sequence D: (10, 12, . . . , 20)

The `\ldots` macro behaves the same as the `\dots` macro.

D.3.6 Floating Object Format

D.3.6.1 Ruled Line in Table

They say that tables drawn by using ruled lines of plain LaTeX look ugly.[13] Vertical lines should be avoided and horizontal lines should be used sparingly, especially in tables of simple structure.

Table D.3 (corresponding to a table from a now-deleted section) is drawn by using the features of "booktabs" and "xcolor" packages. Note that ruled lines of booktabs can not be mixed with vertical lines in a table.[14]

Table D.3: Refrigeration Power Consumption

Situation	T (K)	C_P	Power per watt waste heat (W)
Dry Ice	195	1.990	0.5
Liquid N_2	77	0.356	2.8
Liquid H_2	20	0.073	13.7
Liquid He	4	0.0138	72.3
IBM Q	0.015	0.000051	19,500.0

D.3.6.2 Position of Caption

In LaTeX conventions, captions of tables are usually placed above them. The reason is the flow of your eye movement when you look at them. Most tables have a row of heading at the top. You naturally look at the top of a table at first. Captions at the bottom of tables disturb this flow. The

[11] This rule assumes that math mode uses the same upright glyph as text mode. Our default font choice meets the assumption.

[12] To be exact, it is the `\textellipsis` macro that is redefined. The behavior of `\dots` macro in math mode is not affected. The "amsmath" package has another definition of `\dots`. It is not used in perfbook at the moment.

[13] `https://www.inf.ethz.ch/personal/markusp/teaching/guides/guide-tables.pdf`

[14] There is another package named "arydshln" which provides dashed lines to be used in tables. A couple of experimental examples are presented in Appendix D.3.7.2.

www.dbooks.org

same can be said of code snippets, which are read from top to bottom.

For code snippets, the "ruled" style chosen for listing environment places the caption at the top. See Listing D.2 for an example.

As for tables, the position of caption is tweaked by `\floatstyle{}` and `\restylefloat{}` macros in preamble.

Vertical skips below captions are reduced by setting a smaller value to the `\abovecaptionskip` variable, which would also affect captions to figures.

In the tables which use horizontal rules of "booktabs" package, the vertical skips between captions and tables are further reduced by setting a negative value to the `\abovetopsep` variable, which controls the behavior of `\toprule`.

D.3.7 Improvement Candidates

There are a few areas yet to be attempted in perfbook which would further improve its appearance. This section lists such candidates.

D.3.7.1 Grouping Related Figures/Listings

To prevent a pair of closely related figures or listings from being placed in different pages, it is desirable to group them into a single floating object. The "subfig" package provides the features to do so.[15]

Two floating objects can be placed side by side by using `\parbox` or `minipage`. For example, Figures 14.10 and 14.11 can be grouped together by using a pair of `minipage`s as shown in Figures D.1 and D.2.

By using subfig package, Listings 15.4 and 15.5 can be grouped together as shown in Listing D.6 with sub-captions (with a minor change of blank line).

Note that they can not be grouped in the same way as Figures D.1 and D.2 because the "ruled" style prevents their captions from being properly typeset.

The sub-caption can be cited by combining a `\ref{}` macro and a `\subref{}` macro, for example, "Listing D.6 (a)".

It can also be cited by a `\ref{}` macro, for example, "Listing D.6b". Note the difference in the resulting format. For the citing by a `\ref{}` to work, you need to place the `\label{}` macro of the combined floating object ahead of the definition of subfloats. Otherwise, the resulting caption number would be off by one from the actual number.

[15] One problem of grouping figures might be the complexity in LaTeX source.

Table D.4: CPU 0 View of Synchronization Mechanisms on 8-Socket System With Intel Xeon Platinum 8176 CPUs @ 2.10GHz

Operation	Cost (ns)	Ratio (cost/clock)
Clock period	0.5	1.0
Best-case CAS	7.0	14.6
Best-case lock	15.4	32.3
Blind CAS	7.2	15.2
CAS	18.0	37.7
Blind CAS (off-core)	47.5	99.8
CAS (off-core)	101.9	214.0
Blind CAS (off-socket)	148.8	312.5
CAS (off-socket)	442.9	930.1
Comms Fabric	5,000	10,500
Global Comms	195,000,000	409,500,000

D.3.7.2 Table Layout Experiment

This section presents some experimental tables using booktabs, xcolors, and arydshln packages. The corresponding tables in the text have been converted using one of the format shown here. The source of this section can be regarded as a reference to be consulted when new tables are added in the text.

In Table D.4 (corresponding to Table 3.1), the "S" column specifiers provided by the "siunitx" package are used to align numbers.

Table D.5 (corresponding to Table 13.1) is an example of table with a complex header. In Table D.5, the gap in the mid-rule corresponds to the distinction which had been represented by double vertical rules before the conversion. The legends in the frame box appended here explain the abbreviations used in the matrix. Two types of memory barrier are denoted by subscripts here. The legends and subscripts are not present in Table 13.1 since they are redundant there.

Table D.6 (corresponding to Table C.1) is a sequence diagram drawn as a table.

Table D.7 is a tweaked version of Table 9.2. Here, the "Category" column in the original is removed and the categories are indicated in rows of bold-face font just below the mid-rules. This change makes it easier for `\rowcolors{}` command of "xcolor" package to work properly.



Figure D.1: Timer Wheel at 1 kHz



Figure D.2: Timer Wheel at 100 kHz

Listing D.6: Message-Passing Litmus Test (by subfig)

(a) Not Enforcing Order

```
1  C C-MP+o-wmb-o+o-o.litmus
2
3  {
4  }
5
6  P0(int* x0, int* x1) {
7
8    WRITE_ONCE(*x0, 2);
9    smp_wmb();
10   WRITE_ONCE(*x1, 2);
11
12 }
13
14 P1(int* x0, int* x1) {
15
16   int r2;
17   int r3;
18
19   r2 = READ_ONCE(*x1);
20   r3 = READ_ONCE(*x0);
21
22 }
23
24
25 exists (1:r2=2 /\ 1:r3=0)
```

(b) Enforcing Order

```
1  C C-MP+o-wmb-o+o-rmb-o.litmus
2
3  {
4  }
5
6  P0(int* x0, int* x1) {
7
8    WRITE_ONCE(*x0, 2);
9    smp_wmb();
10   WRITE_ONCE(*x1, 2);
11
12 }
13
14 P1(int* x0, int* x1) {
15
16   int r2;
17   int r3;
18
19   r2 = READ_ONCE(*x1);
20   smp_rmb();
21   r3 = READ_ONCE(*x0);
22
23 }
24
25 exists (1:r2=2 /\ 1:r3=0)
```

Table D.8 is another version which keeps original columns and colors rows only where a category has multiple rows. This is done by combining `\rowcolors{}` of "xcolor" and `\cellcolor{}` commands of the "colortbl" package (`\cellcolor{}` overrides `\rowcolors{}`).

In Table 9.2, the latter layout without partial row coloring has been chosen for simplicity.

Table D.9 (corresponding to Table 15.1) is also a sequence diagram drawn as a tabular object.

Table D.10 shows another version of Table D.3 with dashed horizontal and vertical rules of the arydshln package.

Table D.10: Refrigeration Power Consumption

Situation	T (K)	C_P	Power per watt waste heat (W)
Dry Ice	195	1.990	0.5
Liquid N_2	77	0.356	2.8
Liquid H_2	20	0.073	13.7
Liquid He	4	0.0138	72.3
IBM Q	0.015	0.000051	19,500.0

In this case, the vertical dashed rules seems unnecessary. The one without the vertical rules is shown in Table D.11.

www.dbooks.org

Table D.5: Synchronization and Reference Counting

Acquisition	Release			
	Locks	Reference Counts	Hazard Pointers	RCU
Locks	–	CAM_R	M	CA
Reference Counts	A	AM_R	M	A
Hazard Pointers	M	M	M	M
RCU	CA	M_ACA	M	CA

Key:
- A: Atomic counting
- C: Check combined with the atomic acquisition operation
- M: Full memory barriers required
- M_R: Memory barriers required only on release
- M_A: Memory barriers required on acquire

Table D.11: Refrigeration Power Consumption

Situation	T (K)	C_P	Power per watt waste heat (W)
Dry Ice	195	1.990	0.5
Liquid N_2	77	0.356	2.8
Liquid H_2	20	0.073	13.7
Liquid He	4	0.0138	72.3
IBM Q	0.015	0.000051	19,500.0

D.3.7.3 Miscellaneous Candidates

Other improvement candidates are listed in the source of this section as comments.

Table D.6: Cache Coherence Example

			CPU Cache				Memory	
Sequence #	CPU #	Operation	0	1	2	3	0	8
0		Initial State	–/I	–/I	–/I	–/I	V	V
1	0	Load	0/S	–/I	–/I	–/I	V	V
2	3	Load	0/S	–/I	–/I	0/S	V	V
3	0	Invalidation	8/S	–/I	–/I	0/S	V	V
4	2	RMW	8/S	–/I	0/E	–/I	V	V
5	2	Store	8/S	–/I	0/M	–/I	I	V
6	1	Atomic Inc	8/S	0/M	–/I	–/I	I	V
7	1	Writeback	8/S	8/S	–/I	–/I	V	V

Table D.7: RCU Publish-Subscribe and Version Maintenance APIs

Primitives	Availability	Overhead
List traversal		
`list_for_each_entry_rcu()`	2.5.59	Simple instructions (memory barrier on Alpha)
List update		
`list_add_rcu()`	2.5.44	Memory barrier
`list_add_tail_rcu()`	2.5.44	Memory barrier
`list_del_rcu()`	2.5.44	Simple instructions
`list_replace_rcu()`	2.6.9	Memory barrier
`list_splice_init_rcu()`	2.6.21	Grace-period latency
Hlist traversal		
`hlist_for_each_entry_rcu()`	2.6.8	Simple instructions (memory barrier on Alpha)
Hlist update		
`hlist_add_after_rcu()`	2.6.14	Memory barrier
`hlist_add_before_rcu()`	2.6.14	Memory barrier
`hlist_add_head_rcu()`	2.5.64	Memory barrier
`hlist_del_rcu()`	2.5.64	Simple instructions
`hlist_replace_rcu()`	2.6.15	Memory barrier
Pointer traversal		
`rcu_dereference()`	2.6.9	Simple instructions (memory barrier on Alpha)
Pointer update		
`rcu_assign_pointer()`	2.6.10	Memory barrier

www.dbooks.org

Table D.8: RCU Publish-Subscribe and Version Maintenance APIs

Category	Primitives	Availability	Overhead
List traversal	`list_for_each_entry_rcu()`	2.5.59	Simple instructions (memory barrier on Alpha)
List update	`list_add_rcu()`	2.5.44	Memory barrier
	`list_add_tail_rcu()`	2.5.44	Memory barrier
	`list_del_rcu()`	2.5.44	Simple instructions
	`list_replace_rcu()`	2.6.9	Memory barrier
	`list_splice_init_rcu()`	2.6.21	Grace-period latency
Hlist traversal	`hlist_for_each_entry_rcu()`	2.6.8	Simple instructions (memory barrier on Alpha)
Hlist update	`hlist_add_after_rcu()`	2.6.14	Memory barrier
	`hlist_add_before_rcu()`	2.6.14	Memory barrier
	`hlist_add_head_rcu()`	2.5.64	Memory barrier
	`hlist_del_rcu()`	2.5.64	Simple instructions
	`hlist_replace_rcu()`	2.6.15	Memory barrier
Pointer traversal	`rcu_dereference()`	2.6.9	Simple instructions (memory barrier on Alpha)
Pointer update	`rcu_assign_pointer()`	2.6.10	Memory barrier

Table D.9: Memory Misordering: Store-Buffering Sequence of Events

	CPU 0			CPU 1		
	Instruction	Store Buffer	Cache	Instruction	Store Buffer	Cache
1	(Initial state)		`x1==0`	(Initial state)		`x0==0`
2	`x0 = 2;`	`x0==2`	`x1==0`	`x1 = 2;`	`x1==2`	`x0==0`
3	`r2 = x1;` (0)	`x0==2`	`x1==0`	`r2 = x0;` (0)	`x1==2`	`x0==0`
4	(Read-invalidate)	`x0==2`	`x0==0`	(Read-invalidate)	`x1==2`	`x1==0`
5	(Finish store)		`x0==2`	(Finish store)		`x1==2`

The Answer to the Ultimate Question of Life, The Universe, and Everything.

"The Hitchhikers Guide to the Galaxy", Douglas Adams

Appendix E

Answers to Quick Quizzes

E.1 How To Use This Book

Quick Quiz 1.1:
Where are the answers to the Quick Quizzes found? ■

Answer:
In Appendix E starting on page 437. Hey, I thought I owed you an easy one! ❑

Quick Quiz 1.2:
Some of the Quick Quiz questions seem to be from the viewpoint of the reader rather than the author. Is that really the intent? ■

Answer:
Indeed it is! Many are questions that Paul E. McKenney would probably have asked if he was a novice student in a class covering this material. It is worth noting that Paul was taught most of this material by parallel hardware and software, not by professors. In Paul's experience, professors are much more likely to provide answers to verbal questions than are parallel systems, recent advances in voice-activate assistants notwithstanding. Of course, we could have a lengthy debate over which of professors or parallel systems provide the most useful answers to these sorts of questions, but for the time being let's just agree that usefulness of answers varies widely across the population both of professors and of parallel systems.

Other quizzes are quite similar to actual questions that have been asked during conference presentations and lectures covering the material in this book. A few others are from the viewpoint of the author. ❑

Quick Quiz 1.3:
These Quick Quizzes are just not my cup of tea. What can I do about it? ■

Answer:
Here are a few possible strategies:

1. Just ignore the Quick Quizzes and read the rest of the book. You might miss out on the interesting material in some of the Quick Quizzes, but the rest of the book has lots of good material as well. This is an eminently reasonable approach if your main goal is to gain a general understanding of the material or if you are skimming through the book to find a solution to a specific problem.

2. Look at the answer immediately rather than investing a large amount of time in coming up with your own answer. This approach is reasonable when a given Quick Quiz's answer holds the key to a specific problem you are trying to solve. This approach is also reasonable if you want a somewhat deeper understanding of the material, but when you do not expect to be called upon to generate parallel solutions given only a blank sheet of paper.

3. If you find the Quick Quizzes distracting but impossible to ignore, you can always clone the LaTeX source for this book from the git archive. You can then run the command `make nq`, which will produce a `perfbook-nq.pdf`. This PDF contains unobtrusive boxed tags where the Quick Quizzes would otherwise be, and gathers each chapter's Quick Quizzes and their answers at the end of that chapter.

4. Learn to like (or at least tolerate) the Quick Quizzes. Experience indicates that quizzing yourself periodically while reading greatly increases comprehension and depth of understanding.

Note that as of mid-2016 the quick quizzes are hyperlinked to the answers and vice versa. Click either the

Edition.2-rc11

www.dbooks.org

"Quick Quiz" heading or the small black square to move to the beginning of the answer. From the answer, click on the heading or the small black square to move to the beginning of the quiz, or, alternatively, click on the small white square at the end of the answer to move to the end of the corresponding quiz. ❑

E.2 Introduction

Quick Quiz 2.1:
Come on now!!! Parallel programming has been known to be exceedingly hard for many decades. You seem to be hinting that it is not so hard. What sort of game are you playing? ■

Answer:
If you really believe that parallel programming is exceedingly hard, then you should have a ready answer to the question "Why is parallel programming hard?" One could list any number of reasons, ranging from deadlocks to race conditions to testing coverage, but the real answer is that *it is not really all that hard*. After all, if parallel programming was really so horribly difficult, how could a large number of open-source projects, ranging from Apache to MySQL to the Linux kernel, have managed to master it?

A better question might be: "Why is parallel programming *perceived* to be so difficult?" To see the answer, let's go back to the year 1991. Paul McKenney was walking across the parking lot to Sequent's benchmarking center carrying six dual-80486 Sequent Symmetry CPU boards, when he suddenly realized that he was carrying several times the price of the house he had just purchased.[1] This high cost of parallel systems meant that parallel programming was restricted to a privileged few who worked for an employer who either manufactured or could afford to purchase machines costing upwards of $100,000—in 1991 dollars US.

In contrast, in 2020, Paul finds himself typing these words on a six-core x86 laptop. Unlike the dual-80486 CPU boards, this laptop also contains 64 GB of main memory, a 1 TB solid-state disk, a display, Ethernet, USB ports, wireless, and Bluetooth. And the laptop is more than an order of magnitude cheaper than even one of those dual-80486 CPU boards, even before taking inflation into account.

[1] Yes, this sudden realization *did* cause him to walk quite a bit more carefully. Why do you ask?

Parallel systems have truly arrived. They are no longer the sole domain of a privileged few, but something available to almost everyone.

The earlier restricted availability of parallel hardware is the *real* reason that parallel programming is considered so difficult. After all, it is quite difficult to learn to program even the simplest machine if you have no access to it. Since the age of rare and expensive parallel machines is for the most part behind us, the age during which parallel programming is perceived to be mind-crushingly difficult is coming to a close.[2] ❑

Quick Quiz 2.2:
How could parallel programming *ever* be as easy as sequential programming? ■

Answer:
It depends on the programming environment. SQL [Int92] is an underappreciated success story, as it permits programmers who know nothing about parallelism to keep a large parallel system productively busy. We can expect more variations on this theme as parallel computers continue to become cheaper and more readily available. For example, one possible contender in the scientific and technical computing arena is MATLAB*P, which is an attempt to automatically parallelize common matrix operations.

Finally, on Linux and UNIX systems, consider the following shell command:

```
get_input | grep "interesting" | sort
```

This shell pipeline runs the `get_input`, `grep`, and `sort` processes in parallel. There, that wasn't so hard, now was it?

In short, parallel programming is just as easy as sequential programming—at least in those environments that hide the parallelism from the user! ❑

Quick Quiz 2.3:
Oh, really??? What about correctness, maintainability, robustness, and so on? ■

Answer:
These are important goals, but they are just as important for sequential programs as they are for parallel programs. Therefore, important though they are, they do not belong on a list specific to parallel programming. ❑

[2] Parallel programming is in some ways more difficult than sequential programming, for example, parallel validation is more difficult. But no longer mind-crushingly difficult.

Quick Quiz 2.4:
And if correctness, maintainability, and robustness don't make the list, why do productivity and generality? ■

Answer:
Given that parallel programming is perceived to be much harder than sequential programming, productivity is tantamount and therefore must not be omitted. Furthermore, high-productivity parallel-programming environments such as SQL serve a specific purpose, hence generality must also be added to the list. ❑

Quick Quiz 2.5:
Given that parallel programs are much harder to prove correct than are sequential programs, again, shouldn't correctness *really* be on the list? ■

Answer:
From an engineering standpoint, the difficulty in proving correctness, either formally or informally, would be important insofar as it impacts the primary goal of productivity. So, in cases where correctness proofs are important, they are subsumed under the "productivity" rubric. ❑

Quick Quiz 2.6:
What about just having fun? ■

Answer:
Having fun is important as well, but, unless you are a hobbyist, would not normally be a *primary* goal. On the other hand, if you *are* a hobbyist, go wild! ❑

Quick Quiz 2.7:
Are there no cases where parallel programming is about something other than performance? ■

Answer:
There certainly are cases where the problem to be solved is inherently parallel, for example, Monte Carlo methods and some numerical computations. Even in these cases, however, there will be some amount of extra work managing the parallelism.

Parallelism is also sometimes used for reliability. For but one example, triple-modulo redundancy has three systems run in parallel and vote on the result. In extreme cases, the three systems will be independently implemented using different algorithms and technologies. ❑

Quick Quiz 2.8:
Why not instead rewrite programs from inefficient scripting languages to C or C++? ■

Answer:
If the developers, budget, and time is available for such a rewrite, and if the result will attain the required levels of performance on a single CPU, this can be a reasonable approach. ❑

Quick Quiz 2.9:
Why all this prattling on about non-technical issues??? And not just *any* non-technical issue, but *productivity* of all things? Who cares? ■

Answer:
If you are a pure hobbyist, perhaps you don't need to care. But even pure hobbyists will often care about how much they can get done, and how quickly. After all, the most popular hobbyist tools are usually those that are the best suited for the job, and an important part of the definition of "best suited" involves productivity. And if someone is paying you to write parallel code, they will very likely care deeply about your productivity. And if the person paying you cares about something, you would be most wise to pay at least some attention to it!

Besides, if you *really* didn't care about productivity, you would be doing it by hand rather than using a computer! ❑

Quick Quiz 2.10:
Given how cheap parallel systems have become, how can anyone afford to pay people to program them? ■

Answer:
There are a number of answers to this question:

1. Given a large computational cluster of parallel machines, the aggregate cost of the cluster can easily justify substantial developer effort, because the development cost can be spread over the large number of machines.

2. Popular software that is run by tens of millions of users can easily justify substantial developer effort, as the cost of this development can be spread over the tens of millions of users. Note that this includes things like kernels and system libraries.

www.dbooks.org

3. If the low-cost parallel machine is controlling the operation of a valuable piece of equipment, then the cost of this piece of equipment might easily justify substantial developer effort.

4. If the software for the low-cost parallel machine produces an extremely valuable result (e.g., energy savings), then this valuable result might again justify substantial developer cost.

5. Safety-critical systems protect lives, which can clearly justify very large developer effort.

6. Hobbyists and researchers might instead seek knowledge, experience, fun, or glory.

So it is not the case that the decreasing cost of hardware renders software worthless, but rather that it is no longer possible to "hide" the cost of software development within the cost of the hardware, at least not unless there are extremely large quantities of hardware. ❑

Quick Quiz 2.11:
This is a ridiculously unachievable ideal! Why not focus on something that is achievable in practice? ■

Answer:
This is eminently achievable. The cellphone is a computer that can be used to make phone calls and to send and receive text messages with little or no programming or configuration on the part of the end user.

This might seem to be a trivial example at first glance, but if you consider it carefully you will see that it is both simple and profound. When we are willing to sacrifice generality, we can achieve truly astounding increases in productivity. Those who indulge in excessive generality will therefore fail to set the productivity bar high enough to succeed near the top of the software stack. This fact of life even has its own acronym: YAGNI, or "You Ain't Gonna Need It." ❑

Quick Quiz 2.12:
Wait a minute! Doesn't this approach simply shift the development effort from you to whoever wrote the existing parallel software you are using? ■

Answer:
Exactly! And that is the whole point of using existing software. One team's work can be used by many other teams, resulting in a large decrease in overall effort compared to all teams needlessly reinventing the wheel. ❑

Quick Quiz 2.13:
What other bottlenecks might prevent additional CPUs from providing additional performance? ■

Answer:
There are any number of potential bottlenecks:

1. Main memory. If a single thread consumes all available memory, additional threads will simply page themselves silly.

2. Cache. If a single thread's cache footprint completely fills any shared CPU cache(s), then adding more threads will simply thrash those affected caches, as will be seen in Chapter 10.

3. Memory bandwidth. If a single thread consumes all available memory bandwidth, additional threads will simply result in additional queuing on the system interconnect.

4. I/O bandwidth. If a single thread is I/O bound, adding more threads will simply result in them all waiting in line for the affected I/O resource.

Specific hardware systems might have any number of additional bottlenecks. The fact is that every resource which is shared between multiple CPUs or threads is a potential bottleneck. ❑

Quick Quiz 2.14:
Other than CPU cache capacity, what might require limiting the number of concurrent threads? ■

Answer:
There are any number of potential limits on the number of threads:

1. Main memory. Each thread consumes some memory (for its stack if nothing else), so that excessive numbers of threads can exhaust memory, resulting in excessive paging or memory-allocation failures.

2. I/O bandwidth. If each thread initiates a given amount of mass-storage I/O or networking traffic, excessive numbers of threads can result in excessive I/O queuing delays, again degrading performance. Some networking protocols may be subject to timeouts or other failures if there are so many threads that networking events cannot be responded to in a timely fashion.

3. Synchronization overhead. For many synchronization protocols, excessive numbers of threads can result in excessive spinning, blocking, or rollbacks, thus degrading performance.

Specific applications and platforms may have any number of additional limiting factors. ❑

Quick Quiz 2.15:
Just what is "explicit timing"??? ■

Answer:
Where each thread is given access to some set of resources during an agreed-to slot of time. For example, a parallel program with eight threads might be organized into eight-millisecond time intervals, so that the first thread is given access during the first millisecond of each interval, the second thread during the second millisecond, and so on. This approach clearly requires carefully synchronized clocks and careful control of execution times, and therefore should be used with considerable caution.

In fact, outside of hard realtime environments, you almost certainly want to use something else instead. Explicit timing is nevertheless worth a mention, as it is always there when you need it. ❑

Quick Quiz 2.16:
Are there any other obstacles to parallel programming? ■

Answer:
There are a great many other potential obstacles to parallel programming. Here are a few of them:

1. The only known algorithms for a given project might be inherently sequential in nature. In this case, either avoid parallel programming (there being no law saying that your project *has* to run in parallel) or invent a new parallel algorithm.

2. The project allows binary-only plugins that share the same address space, such that no one developer has access to all of the source code for the project. Because many parallel bugs, including deadlocks, are global in nature, such binary-only plugins pose a severe challenge to current software development methodologies. This might well change, but for the time being, all developers of parallel code sharing a given address space need to be able to see *all* of the code running in that address space.

3. The project contains heavily used APIs that were designed without regard to parallelism [AGH+11a, CKZ+13]. Some of the more ornate features of the System V message-queue API form a case in point. Of course, if your project has been around for a few decades, and its developers did not have access to parallel hardware, it undoubtedly has at least its share of such APIs.

4. The project was implemented without regard to parallelism. Given that there are a great many techniques that work extremely well in a sequential environment, but that fail miserably in parallel environments, if your project ran only on sequential hardware for most of its lifetime, then your project undoubtably has at least its share of parallel-unfriendly code.

5. The project was implemented without regard to good software-development practice. The cruel truth is that shared-memory parallel environments are often much less forgiving of sloppy development practices than are sequential environments. You may be well-served to clean up the existing design and code prior to attempting parallelization.

6. The people who originally did the development on your project have since moved on, and the people remaining, while well able to maintain it or add small features, are unable to make "big animal" changes. In this case, unless you can work out a very simple way to parallelize your project, you will probably be best off leaving it sequential. That said, there are a number of simple approaches that you might use to parallelize your project, including running multiple instances of it, using a parallel implementation of some heavily used library function, or making use of some other parallel project, such as a database.

One can argue that many of these obstacles are non-technical in nature, but that does not make them any less real. In short, parallelization of a large body of code can be a large and complex effort. As with any large and complex effort, it makes sense to do your homework beforehand. ❑

E.3 Hardware and its Habits

Quick Quiz 3.1:
Why should parallel programmers bother learning low-

www.dbooks.org

level properties of the hardware? Wouldn't it be easier, better, and more elegant to remain at a higher level of abstraction? ■

Answer:
It might well be easier to ignore the detailed properties of the hardware, but in most cases it would be quite foolish to do so. If you accept that the only purpose of parallelism is to increase performance, and if you further accept that performance depends on detailed properties of the hardware, then it logically follows that parallel programmers are going to need to know at least a few hardware properties.

This is the case in most engineering disciplines. Would *you* want to use a bridge designed by an engineer who did not understand the properties of the concrete and steel making up that bridge? If not, why would you expect a parallel programmer to be able to develop competent parallel software without at least *some* understanding of the underlying hardware? ❑

Quick Quiz 3.2:
What types of machines would allow atomic operations on multiple data elements? ■

Answer:
One answer to this question is that it is often possible to pack multiple elements of data into a single machine word, which can then be manipulated atomically.

A more trendy answer would be machines supporting transactional memory [Lom77, Kni86, HM93]. By early 2014, several mainstream systems provided limited hardware transactional memory implementations, which is covered in more detail in Section 17.3. The jury is still out on the applicability of software transactional memory [MMW07, PW07, RHP+07, CBM+08, DFGG11, MS12], which is covered in Section 17.2. ❑

Quick Quiz 3.3:
So have CPU designers also greatly reduced the overhead of cache misses? ■

Answer:
Unfortunately, not so much. There has been some reduction given constant numbers of CPUs, but the finite speed of light and the atomic nature of matter limits their ability to reduce cache-miss overhead for larger systems. Section 3.3 discusses some possible avenues for possible future progress. ❑

Quick Quiz 3.4:
This is a *simplified* sequence of events? How could it *possibly* be any more complex? ■

Answer:
This sequence ignored a number of possible complications, including:

1. Other CPUs might be concurrently attempting to perform memory-reference operations involving this same cacheline.
2. The cacheline might have been replicated read-only in several CPUs' caches, in which case, it would need to be flushed from their caches.
3. CPU 7 might have been operating on the cache line when the request for it arrived, in which case CPU 7 might need to hold off the request until its own operation completed.
4. CPU 7 might have ejected the cacheline from its cache (for example, in order to make room for other data), so that by the time that the request arrived, the cacheline was on its way to memory.
5. A correctable error might have occurred in the cacheline, which would then need to be corrected at some point before the data was used.

Production-quality cache-coherence mechanisms are extremely complicated due to these sorts of considerations [HP95, CSG99, MHS12, SHW11]. ❑

Quick Quiz 3.5:
Why is it necessary to flush the cacheline from CPU 7's cache? ■

Answer:
If the cacheline was not flushed from CPU 7's cache, then CPUs 0 and 7 might have different values for the same set of variables in the cacheline. This sort of incoherence greatly complicates parallel software, which is why so wise hardware architects avoid it. ❑

Quick Quiz 3.6:
Table 3.1 shows CPU 0 sharing a core with CPU 224. Shouldn't that instead be CPU 1??? ■

Answer:
It is easy to be sympathetic to this view, but the file `/sys/devices/system/cpu/cpu0/cache/`

Table E.1: Performance of Synchronization Mechanisms on 16-CPU 2.8 GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio (cost/clock)
Clock period	0.4	1.0
Same-CPU CAS	12.2	33.8
Same-CPU lock	25.6	71.2
Blind CAS	12.9	35.8
CAS	7.0	19.4
Off-Core		
Blind CAS	31.2	86.6
CAS	31.2	86.5
Off-Socket		
Blind CAS	92.4	256.7
CAS	95.9	266.4
Off-System		
Comms Fabric	2,600	7,220
Global Comms	195,000,000	542,000,000

`index0/shared_cpu_list` really does contain the string `0,224`. Therefore, CPU 0's hyperthread twin really is CPU 224. Some people speculate that this numbering allows naive applications and schedulers to perform better, citing the fact that on many workloads the second hyperthread does not provide a huge amount of additional performance. This speculation assumes that naive applications and schedulers would utilize CPUs in numerical order, leaving aside the weaker hyperthread twin CPUs until all cores are in use. ❑

Quick Quiz 3.7:
Surely the hardware designers could be persuaded to improve this situation! Why have they been content with such abysmal performance for these single-instruction operations? ■

Answer:
The hardware designers *have* been working on this problem, and have consulted with no less a luminary than the late physicist Stephen Hawking. Hawking's observation was that the hardware designers have two basic problems [Gar07]:

1. The finite speed of light, and
2. The atomic nature of matter.

The first problem limits raw speed, and the second limits miniaturization, which in turn limits frequency. And even this sidesteps the power-consumption issue that is currently limiting production frequencies to well below 10 GHz.

In addition, Table 3.1 on page 23 represents a reasonably large system with no fewer 448 hardware threads. Smaller systems often achieve better latency, as may be seen in Table E.1, which represents a much smaller system with only 16 hardware threads. A similar view is provided by the rows of Table 3.1 down to and including the two "Off-core" rows.

Furthermore, newer small-scale single-socket systems such as the laptop on which I am typing this also have more reasonable latencies, as can be seen in Table E.2.

Alternatively, a 64-CPU system in the mid 1990s had cross-interconnect latencies in excess of five microseconds, so even the eight-socket 448-hardware-thread monster shown in Table 3.1 represents more than a five-fold improvement over its 25-years-prior counterparts.

Integration of hardware threads in a single core and multiple cores on a die have improved latencies greatly, at least within the confines of a single core or single die. There has been some improvement in overall system latency, but only by about a factor of two. Unfortunately, neither the speed of light nor the atomic nature of matter has changed much in the past few years [Har16]. Therefore, spatial and temporal locality are first-class concerns for concurrent software, even when running on relatively small systems.

Section 3.3 looks at what else hardware designers might be able to do to ease the plight of parallel programmers. ❑

Quick Quiz 3.8:
These numbers are insanely large! How can I possibly get my head around them? ■

Answer:
Get a roll of toilet paper. In the USA, each roll will normally have somewhere around 350–500 sheets. Tear off one sheet to represent a single clock cycle, setting it aside. Now unroll the rest of the roll.

The resulting pile of toilet paper will likely represent a single CAS cache miss.

For the more-expensive inter-system communications latencies, use several rolls (or multiple cases) of toilet paper to represent the communications latency.

www.dbooks.org

Table E.2: CPU 0 View of Synchronization Mechanisms on 12-CPU Intel Core i7-8750H CPU @ 2.20 GHz

Operation	Cost (ns)	Ratio (cost/clock)	CPUs
Clock period	0.5	1.0	
Same-CPU CAS	6.2	13.6	0
Same-CPU lock	13.5	29.6	0
In-core blind CAS	6.5	14.3	6
In-core CAS	16.2	35.6	6
Off-core blind CAS	22.2	48.8	1–5,7–11
Off-core CAS	53.6	117.9	1–5,7–11
Off-System			
Comms Fabric	5,000	11,000	
Global Comms	195,000,000	429,000,000	

Important safety tip: make sure to account for the needs of those you live with when appropriating toilet paper![3] ❑

Quick Quiz 3.9:
But individual electrons don't move anywhere near that fast, even in conductors!!! The electron drift velocity in a conductor under semiconductor voltage levels is on the order of only one *millimeter* per second. What gives??? ■

Answer:
Electron drift velocity tracks the long-term movement of individual electrons. It turns out that individual electrons bounce around quite randomly, so that their instantaneous speed is very high, but over the long term, they don't move very far. In this, electrons resemble long-distance commuters, who might spend most of their time traveling at full highway speed, but over the long term go nowhere. These commuters' speed might be 70 miles per hour (113 kilometers per hour), but their long-term drift velocity relative to the planet's surface is zero.

Therefore, we should pay attention not to the electrons' drift velocity, but to their instantaneous velocities. However, even their instantaneous velocities are nowhere near a significant fraction of the speed of light. Nevertheless, the measured velocity of electric waves in conductors *is* a substantial fraction of the speed of light, so we still have a mystery on our hands.

The other trick is that electrons interact with each other at significant distances (from an atomic perspective, anyway), courtesy of their negative charge. This interaction is carried out by photons, which *do* move at the speed of light. So even with electricity's electrons, it is photons doing most of the fast footwork.

Extending the commuter analogy, a driver might use a smartphone to inform other drivers of an accident or congestion, thus allowing a change in traffic flow to propagate much faster than the instantaneous velocity of the individual cars. Summarizing the analogy between electricity and traffic flow:

1. The (very low) drift velocity of an electron is similar to the long-term velocity of a commuter, both being very nearly zero.
2. The (still rather low) instantaneous velocity of an electron is similar to the instantaneous velocity of a car in traffic. Both are much higher than the drift velocity, but quite small compared to the rate at which changes propagate.
3. The (much higher) propagation velocity of an electric wave is primarily due to photons transmitting electromagnetic force among the electrons. Similarly, traffic patterns can change quite quickly due to communication among drivers. Not that this is necessarily of much help to the drivers already stuck in traffic, any more than it is to the electrons already pooled in a given capacitor.

Of course, to fully understand this topic, you should read up on electrodynamics. ❑

[3] Especially here in early 2020, in the midst of the coronavirus excitement that is keeping store shelves free of toilet paper and much else besides!

Quick Quiz 3.10:
Given that distributed-systems communication is so horribly expensive, why does anyone bother with such systems? ■

Answer:
There are a number of reasons:

1. Shared-memory multiprocessor systems have strict size limits. If you need more than a few thousand CPUs, you have no choice but to use a distributed system.
2. Large shared-memory systems tend to be more expensive per unit computation than their smaller counterparts.
3. Large shared-memory systems tend to have much longer cache-miss latencies than do smaller system. To see this, compare Table 3.1 on page 23 with Table E.2.
4. The distributed-systems communications operations do not necessarily use much CPU, so that computation can proceed in parallel with message transfer.
5. Many important problems are "embarrassingly parallel", so that extremely large quantities of processing may be enabled by a very small number of messages. SETI@HOME [Uni08b] was but one example of such an application. These sorts of applications can make good use of networks of computers despite extremely long communications latencies.

Thus, large shared-memory systems tend to be used for applications that benefit from faster latencies than can be provided by distributed computing, and particularly for those applications that benefit from a large shared memory.

It is likely that continued work on parallel applications will increase the number of embarrassingly parallel applications that can run well on machines and/or clusters having long communications latencies, reductions in cost being the driving force that it is. That said, greatly reduced hardware latencies would be an extremely welcome development, both for single-system and for distributed computing. ❑

Quick Quiz 3.11:
OK, if we are going to have to apply distributed-programming techniques to shared-memory parallel programs, why not just always use these distributed techniques and dispense with shared memory? ■

Answer:
Because it is often the case that only a small fraction of the program is performance-critical. Shared-memory parallelism allows us to focus distributed-programming techniques on that small fraction, allowing simpler shared-memory techniques to be used on the non-performance-critical bulk of the program. ❑

E.4 Tools of the Trade

Quick Quiz 4.1:
You call these tools??? They look more like low-level synchronization primitives to me! ■

Answer:
They look that way because they are in fact low-level synchronization primitives. And they are in fact the fundamental tools for building low-level concurrent software. ❑

Quick Quiz 4.2:
But this silly shell script isn't a *real* parallel program! Why bother with such trivia??? ■

Answer:
Because you should *never* forget the simple stuff!

Please keep in mind that the title of this book is "Is Parallel Programming Hard, And, If So, What Can You Do About It?". One of the most effective things you can do about it is to avoid forgetting the simple stuff! After all, if you choose to do parallel programming the hard way, you have no one but yourself to blame. ❑

Quick Quiz 4.3:
Is there a simpler way to create a parallel shell script? If so, how? If not, why not? ■

Answer:
One straightforward approach is the shell pipeline:

```
grep $pattern1 | sed -e 's/a/b/' | sort
```

For a sufficiently large input file, `grep` will pattern-match in parallel with `sed` editing and with the input processing of `sort`. See the file `parallel.sh` for a demonstration of shell-script parallelism and pipelining. ❑

www.dbooks.org

Quick Quiz 4.4:
But if script-based parallel programming is so easy, why bother with anything else? ■

Answer:
In fact, it is quite likely that a very large fraction of parallel programs in use today are script-based. However, script-based parallelism does have its limitations:

1. Creation of new processes is usually quite heavyweight, involving the expensive `fork()` and `exec()` system calls.

2. Sharing of data, including pipelining, typically involves expensive file I/O.

3. The reliable synchronization primitives available to scripts also typically involve expensive file I/O.

4. Scripting languages are often too slow, but are often quite useful when coordinating execution of long-running programs written in lower-level programming languages.

These limitations require that script-based parallelism use coarse-grained parallelism, with each unit of work having execution time of at least tens of milliseconds, and preferably much longer.

Those requiring finer-grained parallelism are well advised to think hard about their problem to see if it can be expressed in a coarse-grained form. If not, they should consider using other parallel-programming environments, such as those discussed in Section 4.2. ❑

Quick Quiz 4.5:
Why does this `wait()` primitive need to be so complicated? Why not just make it work like the shell-script `wait` does? ■

Answer:
Some parallel applications need to take special action when specific children exit, and therefore need to wait for each child individually. In addition, some parallel applications need to detect the reason that the child died. As we saw in Listing 4.2, it is not hard to build a `waitall()` function out of the `wait()` function, but it would be impossible to do the reverse. Once the information about a specific child is lost, it is lost. ❑

Quick Quiz 4.6:
Isn't there a lot more to `fork()` and `wait()` than discussed here? ■

Answer:
Indeed there is, and it is quite possible that this section will be expanded in future versions to include messaging features (such as UNIX pipes, TCP/IP, and shared file I/O) and memory mapping (such as `mmap()` and `shmget()`). In the meantime, there are any number of textbooks that cover these primitives in great detail, and the truly motivated can read manpages, existing parallel applications using these primitives, as well as the source code of the Linux-kernel implementations themselves.

It is important to note that the parent process in Listing 4.3 waits until after the child terminates to do its `printf()`. Using `printf()`'s buffered I/O concurrently to the same file from multiple processes is non-trivial, and is best avoided. If you really need to do concurrent buffered I/O, consult the documentation for your OS. For UNIX/Linux systems, Stewart Weiss's lecture notes provide a good introduction with informative examples [Wei13]. ❑

Quick Quiz 4.7:
If the `mythread()` function in Listing 4.4 can simply return, why bother with `pthread_exit()`? ■

Answer:
In this simple example, there is no reason whatsoever. However, imagine a more complex example, where `mythread()` invokes other functions, possibly separately compiled. In such a case, `pthread_exit()` allows these other functions to end the thread's execution without having to pass some sort of error return all the way back up to `mythread()`. ❑

Quick Quiz 4.8:
If the C language makes no guarantees in presence of a data race, then why does the Linux kernel have so many data races? Are you trying to tell me that the Linux kernel is completely broken??? ■

Answer:
Ah, but the Linux kernel is written in a carefully selected superset of the C language that includes special GNU extensions, such as asms, that permit safe execution even in presence of data races. In addition, the Linux kernel does not run on a number of platforms where data races would be especially problematic. For an example, consider

embedded systems with 32-bit pointers and 16-bit busses. On such a system, a data race involving a store to and a load from a given pointer might well result in the load returning the low-order 16 bits of the old value of the pointer concatenated with the high-order 16 bits of the new value of the pointer.

Nevertheless, even in the Linux kernel, data races can be quite dangerous and should be avoided where feasible [Cor12]. ❑

Quick Quiz 4.9:
What if I want several threads to hold the same lock at the same time? ■

Answer:
The first thing you should do is to ask yourself why you would want to do such a thing. If the answer is "because I have a lot of data that is read by many threads, and only occasionally updated", then POSIX reader-writer locks might be what you are looking for. These are introduced in Section 4.2.4.

Another way to get the effect of multiple threads holding the same lock is for one thread to acquire the lock, and then use `pthread_create()` to create the other threads. The question of why this would ever be a good idea is left to the reader. ❑

Quick Quiz 4.10:
Why not simply make the argument to `lock_reader()` on line 6 of Listing 4.5 be a pointer to a `pthread_mutex_t`? ■

Answer:
Because we will need to pass `lock_reader()` to `pthread_create()`. Although we could cast the function when passing it to `pthread_create()`, function casts are quite a bit uglier and harder to get right than are simple pointer casts. ❑

Quick Quiz 4.11:
What is the `READ_ONCE()` on lines 20 and 47 and the `WRITE_ONCE()` on line 47 of Listing 4.5? ■

Answer:
These macros constrain the compiler so as to prevent it from carrying out optimizations that would be problematic for concurrently accessed shared variables. They don't constrain the CPU at all, other than by preventing reordering of accesses to a given single variable. Note that this single-variable constraint does apply to the code shown in Listing 4.5 because only the variable `x` is accessed.

For more information on `READ_ONCE()` and `WRITE_ONCE()`, please see Section 4.2.5. For more information on ordering accesses to multiple variables by multiple threads, please see Chapter 15. In the meantime, `READ_ONCE(x)` has much in common with the GCC intrinsic `__atomic_load_n(&x, __ATOMIC_RELAXED)` and `WRITE_ONCE()` has much in common with the GCC intrinsic `__atomic_store_n(&x, v, __ATOMIC_RELAXED)`. ❑

Quick Quiz 4.12:
Writing four lines of code for each acquisition and release of a `pthread_mutex_t` sure seems painful! Isn't there a better way? ■

Answer:
Indeed! And for that reason, the `pthread_mutex_lock()` and `pthread_mutex_unlock()` primitives are normally wrapped in functions that do this error checking. Later on, we will wrap them with the Linux kernel `spin_lock()` and `spin_unlock()` APIs. ❑

Quick Quiz 4.13:
Is "x = 0" the only possible output from the code fragment shown in Listing 4.6? If so, why? If not, what other output could appear, and why? ■

Answer:
No. The reason that "x = 0" was output was that `lock_reader()` acquired the lock first. Had `lock_writer()` instead acquired the lock first, then the output would have been "x = 3". However, because the code fragment started `lock_reader()` first and because this run was performed on a multiprocessor, one would normally expect `lock_reader()` to acquire the lock first. Nevertheless, there are no guarantees, especially on a busy system. ❑

Quick Quiz 4.14:
Using different locks could cause quite a bit of confusion, what with threads seeing each others' intermediate states. So should well-written parallel programs restrict themselves to using a single lock in order to avoid this kind of confusion? ■

Answer:
Although it is sometimes possible to write a program using a single global lock that both performs and scales well, such programs are exceptions to the rule. You

www.dbooks.org

will normally need to use multiple locks to attain good performance and scalability.

One possible exception to this rule is "transactional memory", which is currently a research topic. Transactional-memory semantics can be loosely thought of as those of a single global lock with optimizations permitted and with the addition of rollback [Boe09]. ❑

Quick Quiz 4.15:
In the code shown in Listing 4.7, is `lock_reader()` guaranteed to see all the values produced by `lock_writer()`? Why or why not? ■

Answer:
No. On a busy system, `lock_reader()` might be preempted for the entire duration of `lock_writer()`'s execution, in which case it would not see *any* of `lock_writer()`'s intermediate states for `x`. ❑

Quick Quiz 4.16:
Wait a minute here!!! Listing 4.6 didn't initialize shared variable `x`, so why does it need to be initialized in Listing 4.7? ■

Answer:
See line 4 of Listing 4.5. Because the code in Listing 4.6 ran first, it could rely on the compile-time initialization of `x`. The code in Listing 4.7 ran next, so it had to re-initialize `x`. ❑

Quick Quiz 4.17:
Instead of using `READ_ONCE()` everywhere, why not just declare `goflag` as `volatile` on line 10 of Listing 4.8? ■

Answer:
A `volatile` declaration is in fact a reasonable alternative in this particular case. However, use of `READ_ONCE()` has the benefit of clearly flagging to the reader that `goflag` is subject to concurrent reads and updates. Note that `READ_ONCE()` is especially useful in cases where most of the accesses are protected by a lock (and thus *not* subject to change), but where a few of the accesses are made outside of the lock. Using a volatile declaration in this case would make it harder for the reader to note the special accesses outside of the lock, and would also make it harder for the compiler to generate good code under the lock. ❑

Quick Quiz 4.18:
`READ_ONCE()` only affects the compiler, not the CPU. Don't we also need memory barriers to make sure that the change in `goflag`'s value propagates to the CPU in a timely fashion in Listing 4.8? ■

Answer:
No, memory barriers are not needed and won't help here. Memory barriers only enforce ordering among multiple memory references: They absolutely do not guarantee to expedite the propagation of data from one part of the system to another.[4] This leads to a quick rule of thumb: You do not need memory barriers unless you are using more than one variable to communicate between multiple threads.

But what about `nreadersrunning`? Isn't that a second variable used for communication? Indeed it is, and there really are the needed memory-barrier instructions buried in `__sync_fetch_and_add()`, which make sure that the thread proclaims its presence before checking to see if it should start. ❑

Quick Quiz 4.19:
Would it ever be necessary to use `READ_ONCE()` when accessing a per-thread variable, for example, a variable declared using GCC's `__thread` storage class? ■

Answer:
It depends. If the per-thread variable was accessed only from its thread, and never from a signal handler, then no. Otherwise, it is quite possible that `READ_ONCE()` is needed. We will see examples of both situations in Section 5.4.4.

This leads to the question of how one thread can gain access to another thread's `__thread` variable, and the answer is that the second thread must store a pointer to its `__thread` pointer somewhere that the first thread has access to. One common approach is to maintain a linked list with one element per thread, and to store the address of each thread's `__thread` variable in the corresponding element. ❑

Quick Quiz 4.20:
Isn't comparing against single-CPU throughput a bit harsh? ■

Answer:
Not at all. In fact, this comparison was, if anything, overly lenient. A more balanced comparison would be

[4] There have been persistent rumors of hardware in which memory barriers actually do expedite propagation of data, but no confirmed sightings.

against single-CPU throughput with the locking primitives commented out. ❑

Quick Quiz 4.21:
But one microsecond is not a particularly small size for a critical section. What do I do if I need a much smaller critical section, for example, one containing only a few instructions? ■

Answer:
If the data being read *never* changes, then you do not need to hold any locks while accessing it. If the data changes sufficiently infrequently, you might be able to checkpoint execution, terminate all threads, change the data, then restart at the checkpoint.

Another approach is to keep a single exclusive lock per thread, so that a thread read-acquires the larger aggregate reader-writer lock by acquiring its own lock, and write-acquires by acquiring all the per-thread locks [HW92]. This can work quite well for readers, but causes writers to incur increasingly large overheads as the number of threads increases.

Some other ways of efficiently handling very small critical sections are described in Chapter 9. ❑

Quick Quiz 4.22:
The system used is a few years old, and new hardware should be faster. So why should anyone worry about reader-writer locks being slow? ■

Answer:
In general, newer hardware is improving. However, it will need to improve several orders of magnitude to permit reader-writer lock to achieve ideal performance on 448 CPUs. Worse yet, the greater the number of CPUs, the larger the required performance improvement. The performance problems of reader-writer locking are therefore very likely to be with us for quite some time to come. ❑

Quick Quiz 4.23:
Is it really necessary to have both sets of primitives? ■

Answer:
Strictly speaking, no. One could implement any member of the second set using the corresponding member of the first set. For example, one could implement `__sync_nand_and_fetch()` in terms of `__sync_fetch_and_nand()` as follows:

```
tmp = v;
ret = __sync_fetch_and_nand(p, tmp);
ret = ~ret & tmp;
```

It is similarly possible to implement `__sync_fetch_and_add()`, `__sync_fetch_and_sub()`, and `__sync_fetch_and_xor()` in terms of their post-value counterparts.

However, the alternative forms can be quite convenient, both for the programmer and for the compiler/library implementor. ❑

Quick Quiz 4.24:
Given that these atomic operations will often be able to generate single atomic instructions that are directly supported by the underlying instruction set, shouldn't they be the fastest possible way to get things done? ■

Answer:
Unfortunately, no. See Chapter 5 for some stark counterexamples. ❑

Quick Quiz 4.25:
What happened to `ACCESS_ONCE()`? ■

Answer:
As of early 2018, the Linux kernel's `ACCESS_ONCE()` is being replaced by `READ_ONCE()` and `WRITE_ONCE()` for reads and writes, respectively [Cor12, Cor14a, Rut17]. `ACCESS_ONCE()` was introduced as a helper in RCU code, but was promoted to core API soon afterward [McK07b, Tor08]. Linux kernel's `READ_ONCE()` and `WRITE_ONCE()` have evolved into complex forms that look quite different than the original `ACCESS_ONCE()` implementation due to the need to support access-once semantics for large structures, but with the possibility of load/store tearing if the structure cannot be loaded and stored with a single machine instruction. ❑

Quick Quiz 4.26:
What happened to the Linux-kernel equivalents to `fork()` and `wait()`? ■

Answer:
They don't really exist. All tasks executing within the Linux kernel share memory, at least unless you want to do a huge amount of memory-mapping work by hand. ❑

www.dbooks.org

Quick Quiz 4.27:
What problems could occur if the variable `counter` were incremented without the protection of `mutex`? ■

Answer:
On CPUs with load-store architectures, incrementing `counter` might compile into something like the following:

```
LOAD counter,r0
INC r0
STORE r0,counter
```

On such machines, two threads might simultaneously load the value of `counter`, each increment it, and each store the result. The new value of `counter` will then only be one greater than before, despite two threads each incrementing it. ❑

Quick Quiz 4.28:
What is wrong with loading Listing 4.14's `global_ptr` up to three times? ■

Answer:
Suppose that `global_ptr` is initially non-`NULL`, but that some other thread sets `global_ptr` to `NULL`. Suppose further that line 1 of the transformed code (Listing 4.15) executes just before `global_ptr` is set to `NULL` and line 2 just after. Then line 1 will conclude that `global_ptr` is non-`NULL`, line 2 will conclude that it is less than `high_address`, so that line 3 passes `do_low()` a `NULL` pointer, which `do_low()` just might not be prepared to deal with.

Your editor made exactly this mistake in the DYNIX/ptx kernel's memory allocator in the early 1990s. Tracking down the bug consumed a holiday weekend not just for your editor, but also for several of his colleagues. In short, this is not a new problem, nor is it likely to go away on its own. ❑

Quick Quiz 4.29:
Why does it matter whether `do_something()` and `do_something_else()` in Listing 4.18 are inline functions? ■

Answer:
Because `gp` is not a static variable, if either `do_something()` or `do_something_else()` were separately compiled, the compiler would have to assume that either or both of these two functions might change the value of `gp`. This possibility would force the compiler to reload `gp` on line 15, thus avoiding the `NULL`-pointer dereference. ❑

Quick Quiz 4.30:
Ouch! So can't the compiler invent a store to a normal variable pretty much any time it likes? ■

Answer:
Thankfully, the answer is no. This is because the compiler is forbidden from introducing data races. The case of inventing a store just before a normal store is quite special: It is not possible for some other entity, be it CPU, thread, signal handler, or interrupt handler, to be able to see the invented store unless the code already has a data race, even without the invented store. And if the code already has a data race, it already invokes the dreaded spectre of undefined behavior, which allows the compiler to generate pretty much whatever code it wants, regardless of the wishes of the developer.

But if the original store is volatile, as in `WRITE_ONCE()`, for all the compiler knows, there might be a side effect associated with the store that could signal some other thread, allowing data-race-free access to the variable. By inventing the store, the compiler might be introducing a data race, which it is not permitted to do.

In the case of `volatile` and atomic variables, the compiler is specifically forbidden from inventing writes. ❑

Quick Quiz 4.31:
But aren't full memory barriers very heavyweight? Isn't there a cheaper way to enforce the ordering needed in Listing 4.29? ■

Answer:
As is often the case, the answer is "it depends". However, if only two threads are accessing the `status` and `other_task_ready` variables, then the `smp_store_release()` and `smp_load_acquire()` functions discussed in Section 4.3.5 will suffice. ❑

Quick Quiz 4.32:
What needs to happen if an interrupt or signal handler might itself be interrupted? ■

Answer:
Then that interrupt handler must follow the same rules that are followed by other interrupted code. Only those handlers that cannot be themselves interrupted or that access no variables shared with an interrupting handler may safely use plain accesses, and even then only if those variables cannot be concurrently accessed by some other CPU or thread. ❑

Quick Quiz 4.33:
How could you work around the lack of a per-thread-variable API on systems that do not provide it? ■

Answer:
One approach would be to create an array indexed by `smp_thread_id()`, and another would be to use a hash table to map from `smp_thread_id()` to an array index—which is in fact what this set of APIs does in pthread environments.

Another approach would be for the parent to allocate a structure containing fields for each desired per-thread variable, then pass this to the child during thread creation. However, this approach can impose large software-engineering costs in large systems. To see this, imagine if all global variables in a large system had to be declared in a single file, regardless of whether or not they were C static variables! ❑

Quick Quiz 4.34:
Wouldn't the shell normally use `vfork()` rather than `fork()`? ■

Answer:
It might well do that, however, checking is left as an exercise for the reader. But in the meantime, I hope that we can agree that `vfork()` is a variant of `fork()`, so that we can use `fork()` as a generic term covering both. ❑

E.5 Counting

Quick Quiz 5.1:
Why should efficient and scalable counting be hard??? After all, computers have special hardware for the sole purpose of doing counting!!! ■

Answer:
Because the straightforward counting algorithms, for example, atomic operations on a shared counter, either are slow and scale badly, or are inaccurate, as will be seen in Section 5.1. ❑

Quick Quiz 5.2:
Network-packet counting problem. Suppose that you need to collect statistics on the number of networking packets transmitted and received. Packets might be transmitted or received by any CPU on the system. Suppose further that your system is capable of handling millions of packets per second per CPU, and that a systems-monitoring package reads the count every five seconds. How would you implement this counter? ■

Answer:
Hint: The act of updating the counter must be blazingly fast, but because the counter is read out only about once in five million updates, the act of reading out the counter can be quite slow. In addition, the value read out normally need not be all that accurate—after all, since the counter is updated a thousand times per millisecond, we should be able to work with a value that is within a few thousand counts of the "true value", whatever "true value" might mean in this context. However, the value read out should maintain roughly the same absolute error over time. For example, a 1 % error might be just fine when the count is on the order of a million or so, but might be absolutely unacceptable once the count reaches a trillion. See Section 5.2. ❑

Quick Quiz 5.3:
Approximate structure-allocation limit problem. Suppose that you need to maintain a count of the number of structures allocated in order to fail any allocations once the number of structures in use exceeds a limit (say, 10,000). Suppose further that the structures are short-lived, the limit is rarely exceeded, and a "sloppy" approximate limit is acceptable. ■

Answer:
Hint: The act of updating the counter must again be blazingly fast, but the counter is read out each time that the counter is increased. However, the value read out need not be accurate *except* that it must distinguish approximately between values below the limit and values greater than or equal to the limit. See Section 5.3. ❑

Quick Quiz 5.4:
Exact structure-allocation limit problem. Suppose that you need to maintain a count of the number of structures allocated in order to fail any allocations once the number of structures in use exceeds an exact limit (again, say 10,000). Suppose further that these structures are short-lived, and that the limit is rarely exceeded, that there is almost always at least one structure in use, and suppose further still that it is necessary to know exactly when this counter reaches zero, for example, in order to free up some memory that is not required unless there is at least one structure in use. ■

www.dbooks.org

Answer:
Hint: The act of updating the counter must once again be blazingly fast, but the counter is read out each time that the counter is increased. However, the value read out need not be accurate *except* that it absolutely must distinguish perfectly between values between the limit and zero on the one hand, and values that either are less than or equal to zero or are greater than or equal to the limit on the other hand. See Section 5.4. ❑

Quick Quiz 5.5:
Removable I/O device access-count problem. Suppose that you need to maintain a reference count on a heavily used removable mass-storage device, so that you can tell the user when it is safe to remove the device. As usual, the user indicates a desire to remove the device, and the system tells the user when it is safe to do so. ■

Answer:
Hint: Yet again, the act of updating the counter must be blazingly fast and scalable in order to avoid slowing down I/O operations, but because the counter is read out only when the user wishes to remove the device, the counter read-out operation can be extremely slow. Furthermore, there is no need to be able to read out the counter at all unless the user has already indicated a desire to remove the device. In addition, the value read out need not be accurate *except* that it absolutely must distinguish perfectly between non-zero and zero values, and even then only when the device is in the process of being removed. However, once it has read out a zero value, it must act to keep the value at zero until it has taken some action to prevent subsequent threads from gaining access to the device being removed. See Section 5.4.6. ❑

Quick Quiz 5.6:
One thing that could be simpler is ++ instead of that concatenation of `READ_ONCE()` and `WRITE_ONCE()`. Why all that extra typing??? ■

Answer:
See Section 4.3.4.1 on page 40 for more information on how the compiler can cause trouble, as well as how `READ_ONCE()` and `WRITE_ONCE()` can avoid this trouble. ❑

Quick Quiz 5.7:
But can't a smart compiler prove that line 5 of Listing 5.1 is equivalent to the ++ operator and produce an x86 add-to-memory instruction? And won't the CPU cache cause this to be atomic? ■

Answer:
Although the ++ operator *could* be atomic, there is no requirement that it be so unless it is applied to a C11 `_Atomic` variable. And indeed, in the absence of `_Atomic`, GCC often chooses to load the value to a register, increment the register, then store the value to memory, which is decidedly non-atomic.

Furthermore, note the volatile casts in `READ_ONCE()` and `WRITE_ONCE()`, which tell the compiler that the location might well be an MMIO device register. Because MMIO registers are not cached, it would be unwise for the compiler to assume that the increment operation is atomic. ❑

Quick Quiz 5.8:
The 8-figure accuracy on the number of failures indicates that you really did test this. Why would it be necessary to test such a trivial program, especially when the bug is easily seen by inspection? ■

Answer:
Not only are there very few trivial parallel programs, and most days I am not so sure that there are many trivial sequential programs, either.

No matter how small or simple the program, if you haven't tested it, it does not work. And even if you have tested it, Murphy's Law says that there will be at least a few bugs still lurking.

Furthermore, while proofs of correctness certainly do have their place, they never will replace testing, including the `counttorture.h` test setup used here. After all, proofs are only as good as the assumptions that they are based on. Finally, proofs can be every bit as buggy as are programs! ❑

Quick Quiz 5.9:
Why doesn't the horizontal dashed line on the x axis meet the diagonal line at $x = 1$? ■

Answer:
Because of the overhead of the atomic operation. The dashed line on the x axis represents the overhead of a single *non-atomic* increment. After all, an *ideal* algorithm would not only scale linearly, it would also incur no performance penalty compared to single-threaded code.

This level of idealism may seem severe, but if it is good enough for Linus Torvalds, it is good enough for you. ❑

Quick Quiz 5.10:
But atomic increment is still pretty fast. And incrementing a single variable in a tight loop sounds pretty unrealistic to me, after all, most of the program's execution should be devoted to actually doing work, not accounting for the work it has done! Why should I care about making this go faster? ■

Answer:
In many cases, atomic increment will in fact be fast enough for you. In those cases, you should by all means use atomic increment. That said, there are many real-world situations where more elaborate counting algorithms are required. The canonical example of such a situation is counting packets and bytes in highly optimized networking stacks, where it is all too easy to find much of the execution time going into these sorts of accounting tasks, especially on large multiprocessors.

In addition, as noted at the beginning of this chapter, counting provides an excellent view of the issues encountered in shared-memory parallel programs. ❑

Quick Quiz 5.11:
But why can't CPU designers simply ship the addition operation to the data, avoiding the need to circulate the cache line containing the global variable being incremented? ■

Answer:
It might well be possible to do this in some cases. However, there are a few complications:

1. If the value of the variable is required, then the thread will be forced to wait for the operation to be shipped to the data, and then for the result to be shipped back.
2. If the atomic increment must be ordered with respect to prior and/or subsequent operations, then the thread will be forced to wait for the operation to be shipped to the data, and for an indication that the operation completed to be shipped back.
3. Shipping operations among CPUs will likely require more lines in the system interconnect, which will consume more die area and more electrical power.

But what if neither of the first two conditions holds? Then you should think carefully about the algorithms discussed in Section 5.2, which achieve near-ideal performance on commodity hardware.

If either or both of the first two conditions hold, there is *some* hope for improved hardware. One could imagine

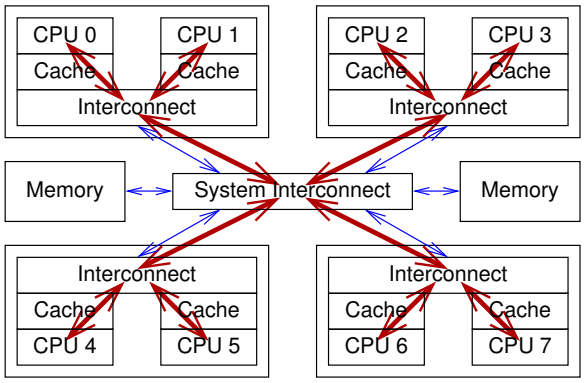


Figure E.1: Data Flow For Global Combining-Tree Atomic Increment

the hardware implementing a combining tree, so that the increment requests from multiple CPUs are combined by the hardware into a single addition when the combined request reaches the hardware. The hardware could also apply an order to the requests, thus returning to each CPU the return value corresponding to its particular atomic increment. This results in instruction latency that varies as $O(\log N)$, where N is the number of CPUs, as shown in Figure E.1. And CPUs with this sort of hardware optimization started to appear in 2011.

This is a great improvement over the $O(N)$ performance of current hardware shown in Figure 5.2, and it is possible that hardware latencies might decrease further if innovations such as three-dimensional fabrication prove practical. Nevertheless, we will see that in some important special cases, software can do *much* better. ❑

Quick Quiz 5.12:
But doesn't the fact that C's "integers" are limited in size complicate things? ■

Answer:
No, because modulo addition is still commutative and associative. At least as long as you use unsigned integers. Recall that in the C standard, overflow of signed integers results in undefined behavior, never mind the fact that machines that do anything other than wrap on overflow are quite rare these days. Unfortunately, compilers frequently carry out optimizations that assume that signed integers will not overflow, so if your code allows signed integers to overflow, you can run into trouble even on modern twos-complement hardware.

That said, one potential source of additional complexity arises when attempting to gather (say) a 64-bit sum

www.dbooks.org

from 32-bit per-thread counters. Dealing with this added complexity is left as an exercise for the reader, for whom some of the techniques introduced later in this chapter could be quite helpful. ❑

Quick Quiz 5.13:
An array??? But doesn't that limit the number of threads? ■

Answer:
It can, and in this toy implementation, it does. But it is not that hard to come up with an alternative implementation that permits an arbitrary number of threads, for example, using GCC's `__thread` facility, as shown in Section 5.2.3. ❑

Quick Quiz 5.14:
What other nasty optimizations could GCC apply? ■

Answer:
See Sections 4.3.4.1 and 15.3 for more information. One nasty optimization would be to apply common subexpression elimination to successive calls to the `read_count()` function, which might come as a surprise to code expecting changes in the values returned from successive calls to that function. ❑

Quick Quiz 5.15:
How does the per-thread `counter` variable in Listing 5.3 get initialized? ■

Answer:
The C standard specifies that the initial value of global variables is zero, unless they are explicitly initialized, thus implicitly initializing all the instances of `counter` to zero. Besides, in the common case where the user is interested only in differences between consecutive reads from statistical counters, the initial value is irrelevant. ❑

Quick Quiz 5.16:
How is the code in Listing 5.3 supposed to permit more than one counter? ■

Answer:
Indeed, this toy example does not support more than one counter. Modifying it so that it can provide multiple counters is left as an exercise to the reader. ❑

Quick Quiz 5.17:
The read operation takes time to sum up the per-thread values, and during that time, the counter could well be changing. This means that the value returned by `read_count()` in Listing 5.3 will not necessarily be exact. Assume that the counter is being incremented at rate r counts per unit time, and that `read_count()`'s execution consumes Δ units of time. What is the expected error in the return value? ■

Answer:
Let's do worst-case analysis first, followed by a less conservative analysis.

In the worst case, the read operation completes immediately, but is then delayed for Δ time units before returning, in which case the worst-case error is simply $r\Delta$.

This worst-case behavior is rather unlikely, so let us instead consider the case where the reads from each of the N counters is spaced equally over the time period Δ. There will be $N+1$ intervals of duration $\frac{\Delta}{N+1}$ between the N reads. The error due to the delay after the read from the last thread's counter will be given by $\frac{r\Delta}{N(N+1)}$, the second-to-last thread's counter by $\frac{2r\Delta}{N(N+1)}$, the third-to-last by $\frac{3r\Delta}{N(N+1)}$, and so on. The total error is given by the sum of the errors due to the reads from each thread's counter, which is:

$$\frac{r\Delta}{N(N+1)} \sum_{i=1}^{N} i \tag{E.1}$$

Expressing the summation in closed form yields:

$$\frac{r\Delta}{N(N+1)} \frac{N(N+1)}{2} \tag{E.2}$$

Canceling yields the intuitively expected result:

$$\frac{r\Delta}{2} \tag{E.3}$$

It is important to remember that error continues accumulating as the caller executes code making use of the count returned by the read operation. For example, if the caller spends time t executing some computation based on the result of the returned count, the worst-case error will have increased to $r(\Delta + t)$.

The expected error will have similarly increased to:

$$r\left(\frac{\Delta}{2} + t\right) \tag{E.4}$$

Of course, it is sometimes unacceptable for the counter to continue incrementing during the read operation. Section 5.4.6 discusses a way to handle this situation.

Thus far, we have been considering a counter that is only increased, never decreased. If the counter value is being changed by r counts per unit time, but in either direction, we should expect the error to reduce. However, the worst case is unchanged because although the counter *could* move in either direction, the worst case is when the read operation completes immediately, but then is delayed for Δ time units, during which time all the changes in the counter's value move it in the same direction, again giving us an absolute error of $r\Delta$.

There are a number of ways to compute the average error, based on a variety of assumptions about the patterns of increments and decrements. For simplicity, let's assume that the f fraction of the operations are decrements, and that the error of interest is the deviation from the counter's long-term trend line. Under this assumption, if f is less than or equal to 0.5, each decrement will be canceled by an increment, so that $2f$ of the operations will cancel each other, leaving $1-2f$ of the operations being uncanceled increments. On the other hand, if f is greater than 0.5, $1-f$ of the decrements are canceled by increments, so that the counter moves in the negative direction by $-1+2(1-f)$, which simplifies to $1-2f$, so that the counter moves an average of $1-2f$ per operation in either case. Therefore, that the long-term movement of the counter is given by $(1-2f)r$. Plugging this into Equation E.3 yields:

$$\frac{(1-2f)r\Delta}{2} \tag{E.5}$$

All that aside, in most uses of statistical counters, the error in the value returned by `read_count()` is irrelevant. This irrelevance is due to the fact that the time required for `read_count()` to execute is normally extremely small compared to the time interval between successive calls to `read_count()`. ❑

Quick Quiz 5.18:
Doesn't that explicit `counterp` array in Listing 5.4 reimpose an arbitrary limit on the number of threads? Why doesn't GCC provide a `per_thread()` interface, similar to the Linux kernel's `per_cpu()` primitive, to allow threads to more easily access each others' per-thread variables? ■

Answer:
Why indeed?

To be fair, GCC faces some challenges that the Linux kernel gets to ignore. When a user-level thread exits, its per-thread variables all disappear, which complicates the problem of per-thread-variable access, particularly before the advent of user-level RCU (see Section 9.5). In contrast, in the Linux kernel, when a CPU goes offline, that CPU's per-CPU variables remain mapped and accessible.

Similarly, when a new user-level thread is created, its per-thread variables suddenly come into existence. In contrast, in the Linux kernel, all per-CPU variables are mapped and initialized at boot time, regardless of whether the corresponding CPU exists yet, or indeed, whether the corresponding CPU will ever exist.

A key limitation that the Linux kernel imposes is a compile-time maximum bound on the number of CPUs, namely, `CONFIG_NR_CPUS`, along with a typically tighter boot-time bound of `nr_cpu_ids`. In contrast, in user space, there is not necessarily a hard-coded upper limit on the number of threads.

Of course, both environments must handle dynamically loaded code (dynamic libraries in user space, kernel modules in the Linux kernel), which increases the complexity of per-thread variables.

These complications make it significantly harder for user-space environments to provide access to other threads' per-thread variables. Nevertheless, such access is highly useful, and it is hoped that it will someday appear.

In the meantime, textbook examples such as this one can use arrays whose limits can be easily adjusted by the user. Alternatively, such arrays can be dynamically allocated and expanded as needed at runtime. Finally, variable-length data structures such as linked lists can be used, as is done in the userspace RCU library [Des09b, DMS+12]. This last approach can also reduce false sharing in some cases. ❑

Quick Quiz 5.19:
Doesn't the check for `NULL` on line 19 of Listing 5.4 add extra branch mispredictions? Why not have a variable set permanently to zero, and point unused counter-pointers to that variable rather than setting them to `NULL`? ■

Answer:
This is a reasonable strategy. Checking for the performance difference is left as an exercise for the reader. However, please keep in mind that the fastpath is not `read_count()`, but rather `inc_count()`. ❑

Quick Quiz 5.20:
Why on earth do we need something as heavyweight as a *lock* guarding the summation in the function `read_count()` in Listing 5.4? ■

www.dbooks.org

Answer:
Remember, when a thread exits, its per-thread variables disappear. Therefore, if we attempt to access a given thread's per-thread variables after that thread exits, we will get a segmentation fault. The lock coordinates summation and thread exit, preventing this scenario.

Of course, we could instead read-acquire a reader-writer lock, but Chapter 9 will introduce even lighter-weight mechanisms for implementing the required coordination.

Another approach would be to use an array instead of a per-thread variable, which, as Alexey Roytman notes, would eliminate the tests against `NULL`. However, array accesses are often slower than accesses to per-thread variables, and use of an array would imply a fixed upper bound on the number of threads. Also, note that neither tests nor locks are needed on the `inc_count()` fastpath. ❑

Quick Quiz 5.21:
Why on earth do we need to acquire the lock in `count_register_thread()` in Listing 5.4? It is a single properly aligned machine-word store to a location that no other thread is modifying, so it should be atomic anyway, right? ■

Answer:
This lock could in fact be omitted, but better safe than sorry, especially given that this function is executed only at thread startup, and is therefore not on any critical path. Now, if we were testing on machines with thousands of CPUs, we might need to omit the lock, but on machines with "only" a hundred or so CPUs, there is no need to get fancy. ❑

Quick Quiz 5.22:
Fine, but the Linux kernel doesn't have to acquire a lock when reading out the aggregate value of per-CPU counters. So why should user-space code need to do this??? ■

Answer:
Remember, the Linux kernel's per-CPU variables are always accessible, even if the corresponding CPU is offline—even if the corresponding CPU never existed and never will exist.

One workaround is to ensure that each thread continues to exist until all threads are finished, as shown in Listing E.1 (`count_tstat.c`). Analysis of this code is left as an exercise to the reader, however, please note that it requires tweaks in the `counttorture.h` counter-evaluation scheme. (Hint: See `#ifndef KEEP_GCC_THREAD_LOCAL`.) Chapter 9 will introduce synchronization mechanisms that handle this situation in a much more graceful manner. ❑

Listing E.1: Per-Thread Statistical Counters With Lockless Summation

```
1  unsigned long __thread counter = 0;
2  unsigned long *counterp[NR_THREADS] = { NULL };
3  int finalthreadcount = 0;
4  DEFINE_SPINLOCK(final_mutex);
5
6  static __inline__ void inc_count(void)
7  {
8    WRITE_ONCE(counter, counter + 1);
9  }
10
11 static __inline__ unsigned long read_count(void)
12                   /* need to tweak counttorture! */
13 {
14   int t;
15   unsigned long sum = 0;
16
17   for_each_thread(t)
18     if (READ_ONCE(counterp[t]) != NULL)
19       sum += READ_ONCE(*counterp[t]);
20   return sum;
21 }
22
23 void count_register_thread(unsigned long *p)
24 {
25   WRITE_ONCE(counterp[smp_thread_id()], &counter);
26 }
27
28 void count_unregister_thread(int nthreadsexpected)
29 {
30   spin_lock(&final_mutex);
31   finalthreadcount++;
32   spin_unlock(&final_mutex);
33   while (READ_ONCE(finalthreadcount) < nthreadsexpected)
34     poll(NULL, 0, 1);
35 }
```

Quick Quiz 5.23:
Why doesn't `inc_count()` in Listing 5.5 need to use atomic instructions? After all, we now have multiple threads accessing the per-thread counters! ■

Answer:
Because one of the two threads only reads, and because the variable is aligned and machine-sized, non-atomic instructions suffice. That said, the `READ_ONCE()` macro is used to prevent compiler optimizations that might otherwise prevent the counter updates from becoming visible to `eventual()`.[5]

An older version of this algorithm did in fact use atomic instructions, kudos to Ersoy Bayramoglu for pointing out that they are in fact unnecessary. However, note that

[5] A simple definition of `READ_ONCE()` is shown in Listing 4.9.

on a 32-bit system, the per-thread `counter` variables might need to be limited to 32 bits in order to sum them accurately, but with a 64-bit `global_count` variable to avoid overflow. In this case, it is necessary to zero the per-thread `counter` variables periodically in order to avoid overflow, which does require atomic instructions. It is extremely important to note that this zeroing cannot be delayed too long or overflow of the smaller per-thread variables will result. This approach therefore imposes real-time requirements on the underlying system, and in turn must be used with extreme care.

In contrast, if all variables are the same size, overflow of any variable is harmless because the eventual sum will be modulo the word size. ❑

Quick Quiz 5.24:
Won't the single global thread in the function `eventual()` of Listing 5.5 be just as severe a bottleneck as a global lock would be? ■

Answer:
In this case, no. What will happen instead is that as the number of threads increases, the estimate of the counter value returned by `read_count()` will become more inaccurate. ❑

Quick Quiz 5.25:
Won't the estimate returned by `read_count()` in Listing 5.5 become increasingly inaccurate as the number of threads rises? ■

Answer:
Yes. If this proves problematic, one fix is to provide multiple `eventual()` threads, each covering its own subset of the other threads. In more extreme cases, a tree-like hierarchy of `eventual()` threads might be required. ❑

Quick Quiz 5.26:
Given that in the eventually-consistent algorithm shown in Listing 5.5 both reads and updates have extremely low overhead and are extremely scalable, why would anyone bother with the implementation described in Section 5.2.2, given its costly read-side code? ■

Answer:
The thread executing `eventual()` consumes CPU time. As more of these eventually-consistent counters are added, the resulting `eventual()` threads will eventually consume all available CPUs. This implementation therefore suffers a different sort of scalability limitation, with the scalability limit being in terms of the number of eventually consistent counters rather than in terms of the number of threads or CPUs.

Of course, it is possible to make other tradeoffs. For example, a single thread could be created to handle all eventually-consistent counters, which would limit the overhead to a single CPU, but would result in increasing update-to-read latencies as the number of counters increased. Alternatively, that single thread could track the update rates of the counters, visiting the frequently-updated counters more frequently. In addition, the number of threads handling the counters could be set to some fraction of the total number of CPUs, and perhaps also adjusted at runtime. Finally, each counter could specify its latency, and deadline-scheduling techniques could be used to provide the required latencies to each counter.

There are no doubt many other tradeoffs that could be made. ❑

Quick Quiz 5.27:
What is the accuracy of the estimate returned by `read_count()` in Listing 5.5? ■

Answer:
A straightforward way to evaluate this estimate is to use the analysis derived in Quick Quiz 5.17, but set Δ to the interval between the beginnings of successive runs of the `eventual()` thread. Handling the case where a given counter has multiple `eventual()` threads is left as an exercise for the reader. ❑

Quick Quiz 5.28:
What fundamental difference is there between counting packets and counting the total number of bytes in the packets, given that the packets vary in size? ■

Answer:
When counting packets, the counter is only incremented by the value one. On the other hand, when counting bytes, the counter might be incremented by largish numbers.

Why does this matter? Because in the increment-by-one case, the value returned will be exact in the sense that the counter must necessarily have taken on that value at some point in time, even if it is impossible to say precisely when that point occurred. In contrast, when counting bytes, two different threads might return values that are inconsistent with any global ordering of operations.

To see this, suppose that thread 0 adds the value three to its counter, thread 1 adds the value five to its counter, and

www.dbooks.org

threads 2 and 3 sum the counters. If the system is "weakly ordered" or if the compiler uses aggressive optimizations, thread 2 might find the sum to be three and thread 3 might find the sum to be five. The only possible global orders of the sequence of values of the counter are 0,3,8 and 0,5,8, and neither order is consistent with the results obtained.

If you missed this one, you are not alone. Michael Scott used this question to stump Paul E. McKenney during Paul's Ph.D. defense. ❑

Quick Quiz 5.29:
Given that the reader must sum all the threads' counters, this counter-read operation could take a long time given large numbers of threads. Is there any way that the increment operation can remain fast and scalable while allowing readers to also enjoy not only reasonable performance and scalability, but also good accuracy? ■

Answer:
One approach would be to maintain a global approximation to the value, similar to the approach described in Section 5.2.4. Updaters would increment their per-thread variable, but when it reached some predefined limit, atomically add it to a global variable, then zero their per-thread variable. This would permit a tradeoff between average increment overhead and accuracy of the value read out. In particular, it would allow sharp bounds on the read-side inaccuracy.

Another approach makes use of the fact that readers often care only about certain transitions in value, not in the exact value. This approach is examined in Section 5.3.

The reader is encouraged to think up and try out other approaches, for example, using a combining tree. ❑

Quick Quiz 5.30:
Why does Listing 5.7 provide `add_count()` and `sub_count()` instead of the `inc_count()` and `dec_count()` interfaces show in Section 5.2? ■

Answer:
Because structures come in different sizes. Of course, a limit counter corresponding to a specific size of structure might still be able to use `inc_count()` and `dec_count()`. ❑

Quick Quiz 5.31:
What is with the strange form of the condition on line 3 of Listing 5.7? Why not the more intuitive form of the fastpath shown in Listing 5.8? ■

Answer:
Two words. "Integer overflow."

Try the formulation in Listing 5.8 with `counter` equal to 10 and `delta` equal to `ULONG_MAX`. Then try it again with the code shown in Listing 5.7.

A good understanding of integer overflow will be required for the rest of this example, so if you have never dealt with integer overflow before, please try several examples to get the hang of it. Integer overflow can sometimes be more difficult to get right than parallel algorithms! ❑

Quick Quiz 5.32:
Why does `globalize_count()` zero the per-thread variables, only to later call `balance_count()` to refill them in Listing 5.7? Why not just leave the per-thread variables non-zero? ■

Answer:
That is in fact what an earlier version of this code did. But addition and subtraction are extremely cheap, and handling all of the special cases that arise is quite complex. Again, feel free to try it yourself, but beware of integer overflow! ❑

Quick Quiz 5.33:
Given that `globalreserve` counted against us in `add_count()`, why doesn't it count for us in `sub_count()` in Listing 5.7? ■

Answer:
The `globalreserve` variable tracks the sum of all threads' `countermax` variables. The sum of these threads' `counter` variables might be anywhere from zero to `globalreserve`. We must therefore take a conservative approach, assuming that all threads' `counter` variables are full in `add_count()` and that they are all empty in `sub_count()`.

But remember this question, as we will come back to it later. ❑

Quick Quiz 5.34:
Suppose that one thread invokes `add_count()` shown in Listing 5.7, and then another thread invokes `sub_count()`. Won't `sub_count()` return failure even though the value of the counter is non-zero? ■

Answer:
Indeed it will! In many cases, this will be a problem, as discussed in Section 5.3.3, and in those cases the algorithms from Section 5.4 will likely be preferable. ❑

Quick Quiz 5.35:
Why have both `add_count()` and `sub_count()` in Listing 5.7? Why not simply pass a negative number to `add_count()`? ■

Answer:
Given that `add_count()` takes an `unsigned long` as its argument, it is going to be a bit tough to pass it a negative number. And unless you have some anti-matter memory, there is little point in allowing negative numbers when counting the number of structures in use!

All kidding aside, it would of course be possible to combine `add_count()` and `sub_count()`, however, the `if` conditions on the combined function would be more complex than in the current pair of functions, which would in turn mean slower execution of these fast paths. ❑

Quick Quiz 5.36:
Why set `counter` to `countermax / 2` in line 15 of Listing 5.9? Wouldn't it be simpler to just take `countermax` counts? ■

Answer:
First, it really is reserving `countermax` counts (see line 14), however, it adjusts so that only half of these are actually in use by the thread at the moment. This allows the thread to carry out at least `countermax / 2` increments or decrements before having to refer back to `globalcount` again.

Note that the accounting in `globalcount` remains accurate, thanks to the adjustment in line 18. ❑

Quick Quiz 5.37:
In Figure 5.6, even though a quarter of the remaining count up to the limit is assigned to thread 0, only an eighth of the remaining count is consumed, as indicated by the uppermost dotted line connecting the center and the rightmost configurations. Why is that? ■

Answer:
The reason this happened is that thread 0's `counter` was set to half of its `countermax`. Thus, of the quarter assigned to thread 0, half of that quarter (one eighth) came from `globalcount`, leaving the other half (again, one eighth) to come from the remaining count.

There are two purposes for taking this approach: (1) To allow thread 0 to use the fastpath for decrements as well as increments, and (2) To reduce the inaccuracies if all threads are monotonically incrementing up towards the limit. To see this last point, step through the algorithm and watch what it does. ❑

Quick Quiz 5.38:
Why is it necessary to atomically manipulate the thread's `counter` and `countermax` variables as a unit? Wouldn't it be good enough to atomically manipulate them individually? ■

Answer:
This might well be possible, but great care is required. Note that removing `counter` without first zeroing `countermax` could result in the corresponding thread increasing `counter` immediately after it was zeroed, completely negating the effect of zeroing the counter.

The opposite ordering, namely zeroing `countermax` and then removing `counter`, can also result in a non-zero `counter`. To see this, consider the following sequence of events:

1. Thread A fetches its `countermax`, and finds that it is non-zero.
2. Thread B zeroes Thread A's `countermax`.
3. Thread B removes Thread A's `counter`.
4. Thread A, having found that its `countermax` is non-zero, proceeds to add to its `counter`, resulting in a non-zero value for `counter`.

Again, it might well be possible to atomically manipulate `countermax` and `counter` as separate variables, but it is clear that great care is required. It is also quite likely that doing so will slow down the fastpath.

Exploring these possibilities are left as exercises for the reader. ❑

Quick Quiz 5.39:
In what way does line 7 of Listing 5.12 violate the C standard? ■

Answer:
It assumes eight bits per byte. This assumption does hold for all current commodity microprocessors that can be easily assembled into shared-memory multiprocessors, but certainly does not hold for all computer systems that have ever run C code. (What could you do instead in order to comply with the C standard? What drawbacks would it have?) ❑

www.dbooks.org

Quick Quiz 5.40:
Given that there is only one `counterandmax` variable, why bother passing in a pointer to it on line 18 of Listing 5.12? ■

Answer:
There is only one `counterandmax` variable *per thread*. Later, we will see code that needs to pass other threads' `counterandmax` variables to `split_counterandmax()`. ❑

Quick Quiz 5.41:
Why does `merge_counterandmax()` in Listing 5.12 return an `int` rather than storing directly into an `atomic_t`? ■

Answer:
Later, we will see that we need the `int` return to pass to the `atomic_cmpxchg()` primitive. ❑

Quick Quiz 5.42:
Yecch! Why the ugly `goto` on line 11 of Listing 5.13? Haven't you heard of the `break` statement??? ■

Answer:
Replacing the `goto` with a `break` would require keeping a flag to determine whether or not line 15 should return, which is not the sort of thing you want on a fastpath. If you really hate the `goto` that much, your best bet would be to pull the fastpath into a separate function that returned success or failure, with "failure" indicating a need for the slowpath. This is left as an exercise for goto-hating readers. ❑

Quick Quiz 5.43:
Why would the `atomic_cmpxchg()` primitive at lines 13–14 of Listing 5.13 ever fail? After all, we picked up its old value on line 9 and have not changed it! ■

Answer:
Later, we will see how the `flush_local_count()` function in Listing 5.15 might update this thread's `counterandmax` variable concurrently with the execution of the fastpath on lines 8–14 of Listing 5.13. ❑

Quick Quiz 5.44:
What stops a thread from simply refilling its `counterandmax` variable immediately after `flush_local_count()` on line 14 of Listing 5.15 empties it? ■

Answer:
This other thread cannot refill its `counterandmax` until the caller of `flush_local_count()` releases the `gblcnt_mutex`. By that time, the caller of `flush_local_count()` will have finished making use of the counts, so there will be no problem with this other thread refilling—assuming that the value of `globalcount` is large enough to permit a refill. ❑

Quick Quiz 5.45:
What prevents concurrent execution of the fastpath of either `add_count()` or `sub_count()` from interfering with the `counterandmax` variable while `flush_local_count()` is accessing it on line 27 of Listing 5.15? ■

Answer:
Nothing. Consider the following three cases:

1. If `flush_local_count()`'s `atomic_xchg()` executes before the `split_counterandmax()` of either fastpath, then the fastpath will see a zero `counter` and `countermax`, and will thus transfer to the slowpath (unless of course `delta` is zero).
2. If `flush_local_count()`'s `atomic_xchg()` executes after the `split_counterandmax()` of either fastpath, but before that fastpath's `atomic_cmpxchg()`, then the `atomic_cmpxchg()` will fail, causing the fastpath to restart, which reduces to case 1 above.
3. If `flush_local_count()`'s `atomic_xchg()` executes after the `atomic_cmpxchg()` of either fastpath, then the fastpath will (most likely) complete successfully before `flush_local_count()` zeroes the thread's `counterandmax` variable.

Either way, the race is resolved correctly. ❑

Quick Quiz 5.46:
Given that the `atomic_set()` primitive does a simple store to the specified `atomic_t`, how can line 21 of `balance_count()` in Listing 5.16 work correctly in face of concurrent `flush_local_count()` updates to this variable? ■

Answer:
The caller of both `balance_count()` and `flush_local_count()` hold `gblcnt_mutex`, so only one may be executing at a given time. ❑

Quick Quiz 5.47:
But signal handlers can be migrated to some other CPU while running. Doesn't this possibility require that atomic instructions and memory barriers are required to reliably communicate between a thread and a signal handler that interrupts that thread? ■

Answer:
No. If the signal handler is migrated to another CPU, then the interrupted thread is also migrated along with it. ❑

Quick Quiz 5.48:
In Figure 5.7, why is the REQ `theft` state colored red? ■

Answer:
To indicate that only the fastpath is permitted to change the `theft` state, and that if the thread remains in this state for too long, the thread running the slowpath will resend the POSIX signal. ❑

Quick Quiz 5.49:
In Figure 5.7, what is the point of having separate REQ and ACK `theft` states? Why not simplify the state machine by collapsing them into a single REQACK state? Then whichever of the signal handler or the fastpath gets there first could set the state to READY. ■

Answer:
Reasons why collapsing the REQ and ACK states would be a very bad idea include:

1. The slowpath uses the REQ and ACK states to determine whether the signal should be retransmitted. If the states were collapsed, the slowpath would have no choice but to send redundant signals, which would have the unhelpful effect of needlessly slowing down the fastpath.

2. The following race would result:

   (a) The slowpath sets a given thread's state to REQACK.

   (b) That thread has just finished its fastpath, and notes the REQACK state.

   (c) The thread receives the signal, which also notes the REQACK state, and, because there is no fastpath in effect, sets the state to READY.

   (d) The slowpath notes the READY state, steals the count, and sets the state to IDLE, and completes.

   (e) The fastpath sets the state to READY, disabling further fastpath execution for this thread.

   The basic problem here is that the combined REQACK state can be referenced by both the signal handler and the fastpath. The clear separation maintained by the four-state setup ensures orderly state transitions.

That said, you might well be able to make a three-state setup work correctly. If you do succeed, compare carefully to the four-state setup. Is the three-state solution really preferable, and why or why not? ❑

Quick Quiz 5.50:
In Listing 5.18's function `flush_local_count_sig()`, why are there `READ_ONCE()` and `WRITE_ONCE()` wrappers around the uses of the `theft` per-thread variable? ■

Answer:
The first one (on line 11) can be argued to be unnecessary. The last two (lines 14 and 16) are important. If these are removed, the compiler would be within its rights to rewrite lines 14–16 as follows:

```
14 theft = THEFT_READY;
15 if (counting) {
16   theft = THEFT_ACK;
17 }
```

This would be fatal, as the slowpath might see the transient value of `THEFT_READY`, and start stealing before the corresponding thread was ready. ❑

Quick Quiz 5.51:
In Listing 5.18, why is it safe for line 28 to directly access the other thread's `countermax` variable? ■

Answer:
Because the other thread is not permitted to change the value of its `countermax` variable unless it holds the `gblcnt_mutex` lock. But the caller has acquired this lock, so it is not possible for the other thread to hold it, and therefore the other thread is not permitted to change its `countermax` variable. We can therefore safely access it—but not change it. ❑

www.dbooks.org

Quick Quiz 5.52:
In Listing 5.18, why doesn't line 33 check for the current thread sending itself a signal? ■

Answer:
There is no need for an additional check. The caller of `flush_local_count()` has already invoked `globalize_count()`, so the check on line 28 will have succeeded, skipping the later `pthread_kill()`. ❑

Quick Quiz 5.53:
The code shown in Listings 5.17 and 5.18 works with GCC and POSIX. What would be required to make it also conform to the ISO C standard? ■

Answer:
The `theft` variable must be of type `sig_atomic_t` to guarantee that it can be safely shared between the signal handler and the code interrupted by the signal. ❑

Quick Quiz 5.54:
In Listing 5.18, why does line 41 resend the signal? ■

Answer:
Because many operating systems over several decades have had the property of losing the occasional signal. Whether this is a feature or a bug is debatable, but irrelevant. The obvious symptom from the user's viewpoint will not be a kernel bug, but rather a user application hanging.

Your user application hanging! ❑

Quick Quiz 5.55:
Not only are POSIX signals slow, sending one to each thread simply does not scale. What would you do if you had (say) 10,000 threads and needed the read side to be fast? ■

Answer:
One approach is to use the techniques shown in Section 5.2.4, summarizing an approximation to the overall counter value in a single variable. Another approach would be to use multiple threads to carry out the reads, with each such thread interacting with a specific subset of the updating threads. ❑

Quick Quiz 5.56:
What if you want an exact limit counter to be exact only for its lower limit, but to allow the upper limit to be inexact? ■

Answer:
One simple solution is to overstate the upper limit by the desired amount. The limiting case of such overstatement results in the upper limit being set to the largest value that the counter is capable of representing. ❑

Quick Quiz 5.57:
What else had you better have done when using a biased counter? ■

Answer:
You had better have set the upper limit to be large enough accommodate the bias, the expected maximum number of accesses, and enough "slop" to allow the counter to work efficiently even when the number of accesses is at its maximum. ❑

Quick Quiz 5.58:
This is ridiculous! We are *read*-acquiring a reader-writer lock to *update* the counter? What are you playing at??? ■

Answer:
Strange, perhaps, but true! Almost enough to make you think that the name "reader-writer lock" was poorly chosen, isn't it? ❑

Quick Quiz 5.59:
What other issues would need to be accounted for in a real system? ■

Answer:
A huge number!

Here are a few to start with:

1. There could be any number of devices, so that the global variables are inappropriate, as are the lack of arguments to functions like `do_io()`.
2. Polling loops can be problematic in real systems, wasting CPU time and energy. In many cases, an event-driven design is far better, for example, where the last completing I/O wakes up the device-removal thread.
3. The I/O might fail, and so `do_io()` will likely need a return value.

4. If the device fails, the last I/O might never complete. In such cases, there might need to be some sort of timeout to allow error recovery.

5. Both `add_count()` and `sub_count()` can fail, but their return values are not checked.

6. Reader-writer locks do not scale well. One way of avoiding the high read-acquisition costs of reader-writer locks is presented in Chapters 7 and 9. ❑

Quick Quiz 5.60:
On the `count_stat.c` row of Table 5.1, we see that the read-side scales linearly with the number of threads. How is that possible given that the more threads there are, the more per-thread counters must be summed up? ■

Answer:
The read-side code must scan the entire fixed-size array, regardless of the number of threads, so there is no difference in performance. In contrast, in the last two algorithms, readers must do more work when there are more threads. In addition, the last two algorithms interpose an additional level of indirection because they map from integer thread ID to the corresponding `__thread` variable. ❑

Quick Quiz 5.61:
Even on the fourth row of Table 5.1, the read-side performance of these statistical counter implementations is pretty horrible. So why bother with them? ■

Answer:
"Use the right tool for the job."

As can be seen from Figure 5.1, single-variable atomic increment need not apply for any job involving heavy use of parallel updates. In contrast, the algorithms shown in the top half of Table 5.1 do an excellent job of handling update-heavy situations. Of course, if you have a read-mostly situation, you should use something else, for example, an eventually consistent design featuring a single atomically incremented variable that can be read out using a single load, similar to the approach used in Section 5.2.4. ❑

Quick Quiz 5.62:
Given the performance data shown in the bottom half of Table 5.1, we should always prefer signals over atomic operations, right? ■

Answer:
That depends on the workload. Note that on a 64-core system, you need more than one hundred non-atomic operations (with roughly a 40-nanosecond performance gain) to make up for even one signal (with almost a 5-*microsecond* performance loss). Although there are no shortage of workloads with far greater read intensity, you will need to consider your particular workload.

In addition, although memory barriers have historically been expensive compared to ordinary instructions, you should check this on the specific hardware you will be running. The properties of computer hardware do change over time, and algorithms must change accordingly. ❑

Quick Quiz 5.63:
Can advanced techniques be applied to address the lock contention for readers seen in the bottom half of Table 5.1? ■

Answer:
One approach is to give up some update-side performance, as is done with scalable non-zero indicators (SNZI) [ELLM07]. There are a number of other ways one might go about this, and these are left as exercises for the reader. Any number of approaches that apply hierarchy, which replace frequent global-lock acquisitions with local lock acquisitions corresponding to lower levels of the hierarchy, should work quite well. ❑

Quick Quiz 5.64:
The ++ operator works just fine for 1,000-digit numbers! Haven't you heard of operator overloading??? ■

Answer:
In the C++ language, you might well be able to use ++ on a 1,000-digit number, assuming that you had access to a class implementing such numbers. But as of 2010, the C language does not permit operator overloading. ❑

Quick Quiz 5.65:
But if we are going to have to partition everything, why bother with shared-memory multithreading? Why not just partition the problem completely and run as multiple processes, each in its own address space? ■

Answer:
Indeed, multiple processes with separate address spaces can be an excellent way to exploit parallelism, as the proponents of the fork-join methodology and the Erlang

www.dbooks.org

language would be very quick to tell you. However, there are also some advantages to shared-memory parallelism:

1. Only the most performance-critical portions of the application must be partitioned, and such portions are usually a small fraction of the application.

2. Although cache misses are quite slow compared to individual register-to-register instructions, they are typically considerably faster than inter-process-communication primitives, which in turn are considerably faster than things like TCP/IP networking.

3. Shared-memory multiprocessors are readily available and quite inexpensive, so, in stark contrast to the 1990s, there is little cost penalty for use of shared-memory parallelism.

As always, use the right tool for the job! ❑

E.6 Partitioning and Synchronization Design

Quick Quiz 6.1:
Is there a better solution to the Dining Philosophers Problem? ■

Answer:
One such improved solution is shown in Figure E.2, where the philosophers are simply provided with an additional five forks. All five philosophers may now eat simultaneously, and there is never any need for philosophers to wait on one another. In addition, this approach offers greatly improved disease control.

This solution might seem like cheating to some, but such "cheating" is key to finding good solutions to many concurrency problems. ❑

Quick Quiz 6.2:
And in just what sense can this "horizontal parallelism" be said to be "horizontal"? ■

Answer:
Inman was working with protocol stacks, which are normally depicted vertically, with the application on top and the hardware interconnect on the bottom. Data flows up and down this stack. "Horizontal parallelism" processes packets from different network connections in parallel, while "vertical parallelism" handles different protocol-processing steps for a given packet in parallel.

"Vertical parallelism" is also called "pipelining". ❑

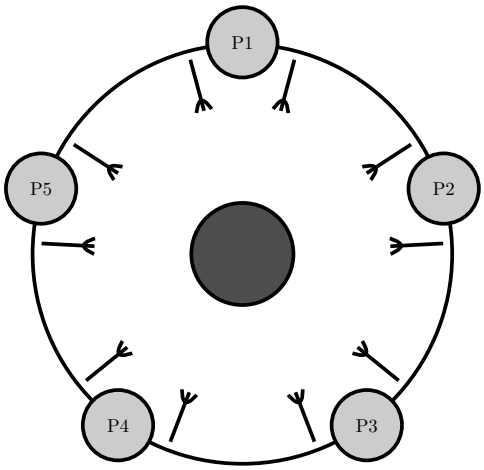


Figure E.2: Dining Philosophers Problem, Fully Partitioned

Quick Quiz 6.3:
In this compound double-ended queue implementation, what should be done if the queue has become non-empty while releasing and reacquiring the lock? ■

Answer:
In this case, simply dequeue an item from the non-empty queue, release both locks, and return. ❑

Quick Quiz 6.4:
Is the hashed double-ended queue a good solution? Why or why not? ■

Answer:
The best way to answer this is to run `lockhdeq.c` on a number of different multiprocessor systems, and you are encouraged to do so in the strongest possible terms. One reason for concern is that each operation on this implementation must acquire not one but two locks.

The first well-designed performance study will be cited.[6] Do not forget to compare to a sequential implementation! ❑

[6] The studies by Dalessandro et al. [DCW+11] and Dice et al. [DLM+10] are excellent starting points.

Quick Quiz 6.5:
Move *all* the elements to the queue that became empty? In what possible universe is this brain-dead solution in any way optimal??? ■

Answer:
It is optimal in the case where data flow switches direction only rarely. It would of course be an extremely poor choice if the double-ended queue was being emptied from both ends concurrently. This of course raises another question, namely, in what possible universe emptying from both ends concurrently would be a reasonable thing to do. Work-stealing queues are one possible answer to this question. ❑

Quick Quiz 6.6:
Why can't the compound parallel double-ended queue implementation be symmetric? ■

Answer:
The need to avoid deadlock by imposing a lock hierarchy forces the asymmetry, just as it does in the fork-numbering solution to the Dining Philosophers Problem (see Section 6.1.1). ❑

Quick Quiz 6.7:
Why is it necessary to retry the right-dequeue operation on line 28 of Listing 6.3? ■

Answer:
This retry is necessary because some other thread might have enqueued an element between the time that this thread dropped `d->rlock` on line 25 and the time that it reacquired this same lock on line 27. ❑

Quick Quiz 6.8:
Surely the left-hand lock must *sometimes* be available!!! So why is it necessary that line 25 of Listing 6.3 unconditionally release the right-hand lock? ■

Answer:
It would be possible to use `spin_trylock()` to attempt to acquire the left-hand lock when it was available. However, the failure case would still need to drop the right-hand lock and then re-acquire the two locks in order. Making this transformation (and determining whether or not it is worthwhile) is left as an exercise for the reader. ❑

Quick Quiz 6.9:
But in the case where data is flowing in only one direction, the algorithm shown in Listing 6.3 will have both ends attempting to acquire the same lock whenever the consuming end empties its underlying double-ended queue. Doesn't that mean that sometimes this algorithm fails to provide concurrent access to both ends of the queue even when the queue contains an arbitrarily large number of elements? ■

Answer:
Indeed it does!

But the same is true of other algorithms claiming this property. For example, in solutions using software transactional memory mechanisms based on hashed arrays of locks, the leftmost and rightmost elements' addresses will sometimes happen to hash to the same lock. These hash collisions will also prevent concurrent access. For another example, solutions using hardware transactional memory mechanisms with software fallbacks [YHLR13, Mer11, JSG12] often use locking within those software fallbacks, and thus suffer (albeit hopefully rarely) from whatever concurrency limitations that these locking solutions suffer from.

Therefore, as of last 2017, all practical solutions to the concurrent double-ended queue problem fail to provide full concurrency in at least some circumstances, including the compound double-ended queue. ❑

Quick Quiz 6.10:
Why are there not one but two solutions to the double-ended queue problem? ■

Answer:
There are actually at least three. The third, by Dominik Dingel, makes interesting use of reader-writer locking, and may be found in `lockrwdeq.c`. ❑

Quick Quiz 6.11:
The tandem double-ended queue runs about twice as fast as the hashed double-ended queue, even when I increase the size of the hash table to an insanely large number. Why is that? ■

Answer:
The hashed double-ended queue's locking design only permits one thread at a time at each end, and further requires two lock acquisitions for each operation. The tandem double-ended queue also permits one thread at a time at each end, and in the common case requires only one lock acquisition per operation. Therefore, the tandem

www.dbooks.org

double-ended queue should be expected to outperform the hashed double-ended queue.

Can you create a double-ended queue that allows multiple concurrent operations at each end? If so, how? If not, why not? ❑

Quick Quiz 6.12:
Is there a significantly better way of handling concurrency for double-ended queues? ■

Answer:
One approach is to transform the problem to be solved so that multiple double-ended queues can be used in parallel, allowing the simpler single-lock double-ended queue to be used, and perhaps also replace each double-ended queue with a pair of conventional single-ended queues. Without such "horizontal scaling", the speedup is limited to 2.0. In contrast, horizontal-scaling designs can achieve very large speedups, and are especially attractive if there are multiple threads working either end of the queue, because in this multiple-thread case the dequeue simply cannot provide strong ordering guarantees. After all, the fact that a given thread removed an item first in no way implies that it will process that item first [HKLP12]. And if there are no guarantees, we may as well obtain the performance benefits that come with refusing to provide these guarantees.

Regardless of whether or not the problem can be transformed to use multiple queues, it is worth asking whether work can be batched so that each enqueue and dequeue operation corresponds to larger units of work. This batching approach decreases contention on the queue data structures, which increases both performance and scalability, as will be seen in Section 6.3. After all, if you must incur high synchronization overheads, be sure you are getting your money's worth.

Other researchers are working on other ways to take advantage of limited ordering guarantees in queues [KLP12]. ❑

Quick Quiz 6.13:
Don't all these problems with critical sections mean that we should just always use non-blocking synchronization [Her90], which don't have critical sections? ■

Answer:
Although non-blocking synchronization can be very useful in some situations, it is no panacea, as discussed in Section 14.2. Also, non-blocking synchronization really does have critical sections, as noted by Josh Triplett. For example, in a non-blocking algorithm based on compare-and-swap operations, the code starting at the initial load and continuing to the compare-and-swap is analogous to a lock-based critical section. ❑

Quick Quiz 6.14:
What are some ways of preventing a structure from being freed while its lock is being acquired? ■

Answer:
Here are a few possible solutions to this *existence guarantee* problem:

1. Provide a statically allocated lock that is held while the per-structure lock is being acquired, which is an example of hierarchical locking (see Section 6.4.2). Of course, using a single global lock for this purpose can result in unacceptably high levels of lock contention, dramatically reducing performance and scalability.

2. Provide an array of statically allocated locks, hashing the structure's address to select the lock to be acquired, as described in Chapter 7. Given a hash function of sufficiently high quality, this avoids the scalability limitations of the single global lock, but in read-mostly situations, the lock-acquisition overhead can result in unacceptably degraded performance.

3. Use a garbage collector, in software environments providing them, so that a structure cannot be deallocated while being referenced. This works very well, removing the existence-guarantee burden (and much else besides) from the developer's shoulders, but imposes the overhead of garbage collection on the program. Although garbage-collection technology has advanced considerably in the past few decades, its overhead may be unacceptably high for some applications. In addition, some applications require that the developer exercise more control over the layout and placement of data structures than is permitted by most garbage collected environments.

4. As a special case of a garbage collector, use a global reference counter, or a global array of reference counters. These have strengths and limitations similar to those called out above for locks.

5. Use *hazard pointers* [Mic04], which can be thought of as an inside-out reference count. Hazard-pointer-

based algorithms maintain a per-thread list of pointers, so that the appearance of a given pointer on any of these lists acts as a reference to the corresponding structure. Hazard pointers are starting to see significant production use (see Section 9.6.3.1).

6. Use transactional memory (TM) [HM93, Lom77, ST95], so that each reference and modification to the data structure in question is performed atomically. Although TM has engendered much excitement in recent years, and seems likely to be of some use in production software, developers should exercise some caution [BLM05, BLM06, MMW07], particularly in performance-critical code. In particular, existence guarantees require that the transaction covers the full path from a global reference to the data elements being updated. For more on TM, including ways to overcome some of its weaknesses by combining it with other synchronization mechanisms, see Sections 17.2 and 17.3.

7. Use RCU, which can be thought of as an extremely lightweight approximation to a garbage collector. Updaters are not permitted to free RCU-protected data structures that RCU readers might still be referencing. RCU is most heavily used for read-mostly data structures, and is discussed at length in Section 9.5.

For more on providing existence guarantees, see Chapters 7 and 9. ❑

Quick Quiz 6.15:
How can a single-threaded 64-by-64 matrix multiple possibly have an efficiency of less than 1.0? Shouldn't all of the traces in Figure 6.17 have efficiency of exactly 1.0 when running on one thread? ■

Answer:
The `matmul.c` program creates the specified number of worker threads, so even the single-worker-thread case incurs thread-creation overhead. Making the changes required to optimize away thread-creation overhead in the single-worker-thread case is left as an exercise to the reader. ❑

Quick Quiz 6.16:
How are data-parallel techniques going to help with matrix multiply? It is *already* data parallel!!! ■

Answer:
I am glad that you are paying attention! This example serves to show that although data parallelism can be a very good thing, it is not some magic wand that automatically wards off any and all sources of inefficiency. Linear scaling at full performance, even to "only" 64 threads, requires care at all phases of design and implementation.

In particular, you need to pay careful attention to the size of the partitions. For example, if you split a 64-by-64 matrix multiply across 64 threads, each thread gets only 64 floating-point multiplies. The cost of a floating-point multiply is minuscule compared to the overhead of thread creation, and cache-miss overhead also plays a role in spoiling the theoretically perfect scalability (and also in making the traces so jagged). The full 448 hardware threads would require a matrix with hundreds of thousands of rows and columns to attain good scalability, but by that point GPGPUs become quite attractive, especially from a price/performance viewpoint.

Moral: If you have a parallel program with variable input, always include a check for the input size being too small to be worth parallelizing. And when it is not helpful to parallelize, it is not helpful to incur the overhead required to spawn a thread, now is it? ❑

Quick Quiz 6.17:
In what situation would hierarchical locking work well? ■

Answer:
If the comparison on line 31 of Listing 6.8 were replaced by a much heavier-weight operation, then releasing `bp->bucket_lock` *might* reduce lock contention enough to outweigh the overhead of the extra acquisition and release of `cur->node_lock`. ❑

Quick Quiz 6.18:
Doesn't this resource-allocator design resemble that of the approximate limit counters covered in Section 5.3? ■

Answer:
Indeed it does! We are used to thinking of allocating and freeing memory, but the algorithms in Section 5.3 are taking very similar actions to allocate and free "count". ❑

Quick Quiz 6.19:
In Figure 6.21, there is a pattern of performance rising with increasing run length in groups of three samples, for example, for run lengths 10, 11, and 12. Why? ■

www.dbooks.org

Answer:
This is due to the per-CPU target value being three. A run length of 12 must acquire the global-pool lock twice, while a run length of 13 must acquire the global-pool lock three times. ❑

Quick Quiz 6.20:
Allocation failures were observed in the two-thread tests at run lengths of 19 and greater. Given the global-pool size of 40 and the per-thread target pool size s of three, number of threads n equal to two, and assuming that the per-thread pools are initially empty with none of the memory in use, what is the smallest allocation run length m at which failures can occur? (Recall that each thread repeatedly allocates m block of memory, and then frees the m blocks of memory.) Alternatively, given n threads each with pool size s, and where each thread repeatedly first allocates m blocks of memory and then frees those m blocks, how large must the global pool size be? *Note:* Obtaining the correct answer will require you to examine the `smpalloc.c` source code, and very likely single-step it as well. You have been warned! ■

Answer:
This solution is adapted from one put forward by Alexey Roytman. It is based on the following definitions:

g Number of blocks globally available.

i Number of blocks left in the initializing thread's per-thread pool. (This is one reason you needed to look at the code!)

m Allocation/free run length.

n Number of threads, excluding the initialization thread.

p Per-thread maximum block consumption, including both the blocks actually allocated and the blocks remaining in the per-thread pool.

The values g, m, and n are given. The value for p is m rounded up to the next multiple of s, as follows:

$$p = s\left\lceil \frac{m}{s} \right\rceil \tag{E.6}$$

The value for i is as follows:

$$i = \begin{cases} g \pmod{2s} = 0 : 2s \\ g \pmod{2s} \neq 0 : g \pmod{2s} \end{cases} \tag{E.7}$$

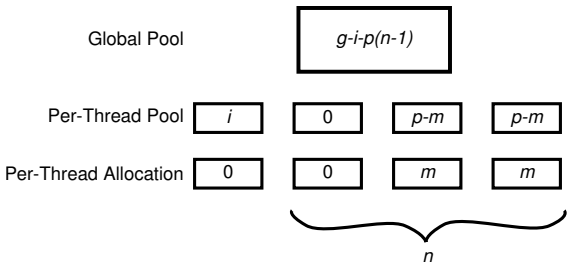


Figure E.3: Allocator Cache Run-Length Analysis

The relationships between these quantities are shown in Figure E.3. The global pool is shown on the top of this figure, and the "extra" initializer thread's per-thread pool and per-thread allocations are the left-most pair of boxes. The initializer thread has no blocks allocated, but has i blocks stranded in its per-thread pool. The rightmost two pairs of boxes are the per-thread pools and per-thread allocations of threads holding the maximum possible number of blocks, while the second-from-left pair of boxes represents the thread currently trying to allocate.

The total number of blocks is g, and adding up the per-thread allocations and per-thread pools, we see that the global pool contains $g - i - p(n-1)$ blocks. If the allocating thread is to be successful, it needs at least m blocks in the global pool, in other words:

$$g - i - p(n-1) \geq m \tag{E.8}$$

The question has $g = 40$, $s = 3$, and $n = 2$. Equation E.7 gives $i = 4$, and Equation E.6 gives $p = 18$ for $m = 18$ and $p = 21$ for $m = 19$. Plugging these into Equation E.8 shows that $m = 18$ will not overflow, but that $m = 19$ might well do so.

The presence of i could be considered to be a bug. After all, why allocate memory only to have it stranded in the initialization thread's cache? One way of fixing this would be to provide a `memblock_flush()` function that flushed the current thread's pool into the global pool. The initialization thread could then invoke this function after freeing all of the blocks. ❑

E.7 Locking

Quick Quiz 7.1:
Just how can serving as a whipping boy be considered to be in any way honorable??? ■

Answer:
The reason locking serves as a research-paper whipping boy is because it is heavily used in practice. In contrast, if no one used or cared about locking, most research papers would not bother even mentioning it. ❑

Quick Quiz 7.2:
But the definition of lock-based deadlock only said that each thread was holding at least one lock and waiting on another lock that was held by some thread. How do you know that there is a cycle? ■

Answer:
Suppose that there is no cycle in the graph. We would then have a directed acyclic graph (DAG), which would have at least one leaf node.

If this leaf node was a lock, then we would have a thread that was waiting on a lock that wasn't held by any thread, counter to the definition. In this case the thread would immediately acquire the lock.

On the other hand, if this leaf node was a thread, then we would have a thread that was not waiting on any lock, again counter to the definition. And in this case, the thread would either be running or be blocked on something that is not a lock. In the first case, in the absence of infinite-loop bugs, the thread will eventually release the lock. In the second case, in the absence of a failure-to-wake bug, the thread will eventually wake up and release the lock.[7]

Therefore, given this definition of lock-based deadlock, there must be a cycle in the corresponding graph. ❑

Quick Quiz 7.3:
Are there any exceptions to this rule, so that there really could be a deadlock cycle containing locks from both the library and the caller, even given that the library code never invokes any of the caller's functions? ■

Answer:
Indeed there are! Here are a few of them:

1. If one of the library function's arguments is a pointer to a lock that this library function acquires, and if the library function holds one of its locks while acquiring the caller's lock, then we could have a deadlock cycle involving both caller and library locks.
2. If one of the library functions returns a pointer to a lock that is acquired by the caller, and if the caller acquires one of its locks while holding the library's lock, we could again have a deadlock cycle involving both caller and library locks.
3. If one of the library functions acquires a lock and then returns while still holding it, and if the caller acquires one of its locks, we have yet another way to create a deadlock cycle involving both caller and library locks.
4. If the caller has a signal handler that acquires locks, then the deadlock cycle can involve both caller and library locks. In this case, however, the library's locks are innocent bystanders in the deadlock cycle. That said, please note that acquiring a lock from within a signal handler is a no-no in most environments—it is not just a bad idea, it is unsupported. But if you absolutely must acquire a lock in a signal handler, be sure to block that signal while holding that same lock in thread context. ❑

Quick Quiz 7.4:
But if `qsort()` releases all its locks before invoking the comparison function, how can it protect against races with other `qsort()` threads? ■

Answer:
By privatizing the data elements being compared (as discussed in Chapter 8) or through use of deferral mechanisms such as reference counting (as discussed in Chapter 9). Or through use of layered locking hierarchies, as described in Section 7.1.1.3.

On the other hand, changing a key in a list that is currently being sorted is at best rather brave. ❑

Quick Quiz 7.5:
Name one common situation where a pointer to a lock is passed into a function. ■

Answer:
Locking primitives, of course! ❑

Quick Quiz 7.6:
Doesn't the fact that `pthread_cond_wait()` first re-

[7] Of course, one type of failure-to-wake bug is a deadlock that involves not only locks, but also non-lock resources. But the question really did say "lock-based deadlock"!

www.dbooks.org

leases the mutex and then re-acquires it eliminate the possibility of deadlock? ■

Answer:
Absolutely not!

Consider a program that acquires `mutex_a`, and then `mutex_b`, in that order, and then passes `mutex_a` to `pthread_cond_wait`. Now, `pthread_cond_wait` will release `mutex_a`, but will re-acquire it before returning. If some other thread acquires `mutex_a` in the meantime and then blocks on `mutex_b`, the program will deadlock. ❑

Quick Quiz 7.7:
Can the transformation from Listing 7.3 to Listing 7.4 be applied universally? ■

Answer:
Absolutely not!

This transformation assumes that the `layer_2_processing()` function is idempotent, given that it might be executed multiple times on the same packet when the `layer_1()` routing decision changes. Therefore, in real life, this transformation can become arbitrarily complex. ❑

Quick Quiz 7.8:
But the complexity in Listing 7.4 is well worthwhile given that it avoids deadlock, right? ■

Answer:
Maybe.

If the routing decision in `layer_1()` changes often enough, the code will always retry, never making forward progress. This is termed "livelock" if no thread makes any forward progress or "starvation" if some threads make forward progress but others do not (see Section 7.1.2). ❑

Quick Quiz 7.9:
When using the "acquire needed locks first" approach described in Section 7.1.1.6, how can livelock be avoided? ■

Answer:
Provide an additional global lock. If a given thread has repeatedly tried and failed to acquire the needed locks, then have that thread unconditionally acquire the new global lock, and then unconditionally acquire any needed locks. (Suggested by Doug Lea.) ❑

Quick Quiz 7.10:
Suppose Lock A is never acquired within a signal handler, but Lock B is acquired both from thread context and by signal handlers. Suppose further that Lock A is sometimes acquired with signals unblocked. Why is it illegal to acquire Lock A holding Lock B? ■

Answer:
Because this would lead to deadlock. Given that Lock A is sometimes held outside of a signal handler without blocking signals, a signal might be handled while holding this lock. The corresponding signal handler might then acquire Lock B, so that Lock B is acquired while holding Lock A. Therefore, if we also acquire Lock A while holding Lock B, we will have a deadlock cycle. Note that this problem exists even if signals are blocked while holding Lock B.

This is another reason to be very careful with locks that are acquired within interrupt or signal handlers. But the Linux kernel's lock dependency checker knows about this situation and many others as well, so please do make full use of it! ❑

Quick Quiz 7.11:
How can you legally block signals within a signal handler? ■

Answer:
One of the simplest and fastest ways to do so is to use the `sa_mask` field of the `struct sigaction` that you pass to `sigaction()` when setting up the signal. ❑

Quick Quiz 7.12:
If acquiring locks in signal handlers is such a bad idea, why even discuss ways of making it safe? ■

Answer:
Because these same rules apply to the interrupt handlers used in operating-system kernels and in some embedded applications.

In many application environments, acquiring locks in signal handlers is frowned upon [Ope97]. However, that does not stop clever developers from (perhaps unwisely) fashioning home-brew locks out of atomic operations. And atomic operations are in many cases perfectly legal in signal handlers. ❑

Quick Quiz 7.13:
Given an object-oriented application that passes control freely among a group of objects such that there is no

straightforward locking hierarchy,[a] layered or otherwise, how can this application be parallelized? ■

[a] Also known as "object-oriented spaghetti code."

Answer:
There are a number of approaches:

1. In the case of parametric search via simulation, where a large number of simulations will be run in order to converge on (for example) a good design for a mechanical or electrical device, leave the simulation single-threaded, but run many instances of the simulation in parallel. This retains the object-oriented design, and gains parallelism at a higher level, and likely also avoids both deadlocks and synchronization overhead.

2. Partition the objects into groups such that there is no need to operate on objects in more than one group at a given time. Then associate a lock with each group. This is an example of a single-lock-at-a-time design, which discussed in Section 7.1.1.7.

3. Partition the objects into groups such that threads can all operate on objects in the groups in some groupwise ordering. Then associate a lock with each group, and impose a locking hierarchy over the groups.

4. Impose an arbitrarily selected hierarchy on the locks, and then use conditional locking if it is necessary to acquire a lock out of order, as was discussed in Section 7.1.1.5.

5. Before carrying out a given group of operations, predict which locks will be acquired, and attempt to acquire them before actually carrying out any updates. If the prediction turns out to be incorrect, drop all the locks and retry with an updated prediction that includes the benefit of experience. This approach was discussed in Section 7.1.1.6.

6. Use transactional memory. This approach has a number of advantages and disadvantages which will be discussed in Sections 17.2–17.3.

7. Refactor the application to be more concurrency-friendly. This would likely also have the side effect of making the application run faster even when single-threaded, but might also make it more difficult to modify the application.

8. Use techniques from later chapters in addition to locking. ❑

Quick Quiz 7.14:
How can the livelock shown in Listing 7.5 be avoided? ■

Answer:
Listing 7.4 provides some good hints. In many cases, livelocks are a hint that you should revisit your locking design. Or visit it in the first place if your locking design "just grew".

That said, one good-and-sufficient approach due to Doug Lea is to use conditional locking as described in Section 7.1.1.5, but combine this with acquiring all needed locks first, before modifying shared data, as described in Section 7.1.1.6. If a given critical section retries too many times, unconditionally acquire a global lock, then unconditionally acquire all the needed locks. This avoids both deadlock and livelock, and scales reasonably assuming that the global lock need not be acquired too often. ❑

Quick Quiz 7.15:
What problems can you spot in the code in Listing 7.6? ■

Answer:
Here are a couple:

1. A one-second wait is way too long for most uses. Wait intervals should begin with roughly the time required to execute the critical section, which will normally be in the microsecond or millisecond range.

2. The code does not check for overflow. On the other hand, this bug is nullified by the previous bug: 32 bits worth of seconds is more than 50 years. ❑

Quick Quiz 7.16:
Wouldn't it be better just to use a good parallel design so that lock contention was low enough to avoid unfairness? ■

Answer:
It would be better in some sense, but there are situations where it can be appropriate to use designs that sometimes result in high lock contentions.

For example, imagine a system that is subject to a rare error condition. It might well be best to have a

www.dbooks.org

simple error-handling design that has poor performance and scalability for the duration of the rare error condition, as opposed to a complex and difficult-to-debug design that is helpful only when one of those rare error conditions is in effect.

That said, it is usually worth putting some effort into attempting to produce a design that both simple as well as efficient during error conditions, for example by partitioning the problem. ❑

Quick Quiz 7.17:
How might the lock holder be interfered with? ■

Answer:
If the data protected by the lock is in the same cache line as the lock itself, then attempts by other CPUs to acquire the lock will result in expensive cache misses on the part of the CPU holding the lock. This is a special case of false sharing, which can also occur if a pair of variables protected by different locks happen to share a cache line. In contrast, if the lock is in a different cache line than the data that it protects, the CPU holding the lock will usually suffer a cache miss only on first access to a given variable.

Of course, the downside of placing the lock and data into separate cache lines is that the code will incur two cache misses rather than only one in the uncontended case. As always, choose wisely! ❑

Quick Quiz 7.18:
Does it ever make sense to have an exclusive lock acquisition immediately followed by a release of that same lock, that is, an empty critical section? ■

Answer:
Empty lock-based critical sections are rarely used, but they do have their uses. The point is that the semantics of exclusive locks have two components: (1) the familiar data-protection semantic and (2) a messaging semantic, where releasing a given lock notifies a waiting acquisition of that same lock. An empty critical section uses the messaging component without the data-protection component.

The rest of this answer provides some example uses of empty critical sections, however, these examples should be considered "gray magic."[8] As such, empty critical sections are almost never used in practice. Nevertheless, pressing on into this gray area . . .

One historical use of empty critical sections appeared in the networking stack of the 2.4 Linux kernel through use of a read-side-scalable reader-writer lock called `brlock` for "big reader lock". This use case is a way of approximating the semantics of read-copy update (RCU), which is discussed in Section 9.5. And in fact this Linux-kernel use case has been replaced with RCU.

The empty-lock-critical-section idiom can also be used to reduce lock contention in some situations. For example, consider a multithreaded user-space application where each thread processes units of work maintained in a per-thread list, where threads are prohibited from touching each others' lists [McK12e]. There could also be updates that require that all previously scheduled units of work have completed before the update can progress. One way to handle this is to schedule a unit of work on each thread, so that when all of these units of work complete, the update may proceed.

In some applications, threads can come and go. For example, each thread might correspond to one user of the application, and thus be removed when that user logs out or otherwise disconnects. In many applications, threads cannot depart atomically: They must instead explicitly unravel themselves from various portions of the application using a specific sequence of actions. One specific action will be refusing to accept further requests from other threads, and another specific action will be disposing of any remaining units of work on its list, for example, by placing these units of work in a global work-item-disposal list to be taken by one of the remaining threads. (Why not just drain the thread's work-item list by executing each item? Because a given work item might generate more work items, so that the list could not be drained in a timely fashion.)

If the application is to perform and scale well, a good locking design is required. One common solution is to have a global lock (call it `G`) protecting the entire process of departing (and perhaps other things as well), with finer-grained locks protecting the individual unraveling operations.

Now, a departing thread must clearly refuse to accept further requests before disposing of the work on its list, because otherwise additional work might arrive after the disposal action, which would render that disposal action ineffective. So simplified pseudocode for a departing thread might be as follows:

1. Acquire lock `G`.
2. Acquire the lock guarding communications.
3. Refuse further communications from other threads.

[8] Thanks to Alexey Roytman for this description.

4. Release the lock guarding communications.
5. Acquire the lock guarding the global work-item-disposal list.
6. Move all pending work items to the global work-item-disposal list.
7. Release the lock guarding the global work-item-disposal list.
8. Release lock G.

Of course, a thread that needs to wait for all pre-existing work items will need to take departing threads into account. To see this, suppose that this thread starts waiting for all pre-existing work items just after a departing thread has refused further communications from other threads. How can this thread wait for the departing thread's work items to complete, keeping in mind that threads are not allowed to access each others' lists of work items?

One straightforward approach is for this thread to acquire G and then the lock guarding the global work-item-disposal list, then move the work items to its own list. The thread then release both locks, places a work item on the end of its own list, and then wait for all of the work items that it placed on each thread's list (including its own) to complete.

This approach does work well in many cases, but if special processing is required for each work item as it is pulled in from the global work-item-disposal list, the result could be excessive contention on G. One way to avoid that contention is to acquire G and then immediately release it. Then the process of waiting for all prior work items look something like the following:

1. Set a global counter to one and initialize a condition variable to zero.
2. Send a message to all threads to cause them to atomically increment the global counter, and then to enqueue a work item. The work item will atomically decrement the global counter, and if the result is zero, it will set a condition variable to one.
3. Acquire G, which will wait on any currently departing thread to finish departing. Because only one thread may depart at a time, all the remaining threads will have already received the message sent in the preceding step.
4. Release G.
5. Acquire the lock guarding the global work-item-disposal list.
6. Move all work items from the global work-item-disposal list to this thread's list, processing them as needed along the way.
7. Release the lock guarding the global work-item-disposal list.
8. Enqueue an additional work item onto this thread's list. (As before, this work item will atomically decrement the global counter, and if the result is zero, it will set a condition variable to one.)
9. Wait for the condition variable to take on the value one.

Once this procedure completes, all pre-existing work items are guaranteed to have completed. The empty critical sections are using locking for messaging as well as for protection of data. ❑

Quick Quiz 7.19:
Is there any other way for the VAX/VMS DLM to emulate a reader-writer lock? ■

Answer:
There are in fact several. One way would be to use the null, protected-read, and exclusive modes. Another way would be to use the null, protected-read, and concurrent-write modes. A third way would be to use the null, concurrent-read, and exclusive modes. ❑

Quick Quiz 7.20:
The code in Listing 7.7 is ridiculously complicated! Why not conditionally acquire a single global lock? ■

Answer:
Conditionally acquiring a single global lock does work very well, but only for relatively small numbers of CPUs. To see why it is problematic in systems with many hundreds of CPUs, look at Figure 5.1. ❑

Quick Quiz 7.21:
Wait a minute! If we "win" the tournament on line 16 of Listing 7.7, we get to do all the work of `do_force_quiescent_state()`. Exactly how is that a win, really? ■

Answer:
How indeed? This just shows that in concurrency, just as

www.dbooks.org

in life, one should take care to learn exactly what winning entails before playing the game. ❑

Quick Quiz 7.22:
Why not rely on the C language's default initialization of zero instead of using the explicit initializer shown on line 2 of Listing 7.8? ■

Answer:
Because this default initialization does not apply to locks allocated as auto variables within the scope of a function. ❑

Quick Quiz 7.23:
Why bother with the inner loop on lines 7–8 of Listing 7.8? Why not simply repeatedly do the atomic exchange operation on line 6? ■

Answer:
Suppose that the lock is held and that several threads are attempting to acquire the lock. In this situation, if these threads all loop on the atomic exchange operation, they will ping-pong the cache line containing the lock among themselves, imposing load on the interconnect. In contrast, if these threads are spinning in the inner loop on lines 7–8, they will each spin within their own caches, placing negligible load on the interconnect. ❑

Quick Quiz 7.24:
Why not simply store zero into the lock word on line 14 of Listing 7.8? ■

Answer:
This can be a legitimate implementation, but only if this store is preceded by a memory barrier and makes use of `WRITE_ONCE()`. The memory barrier is not required when the `xchg()` operation is used because this operation implies a full memory barrier due to the fact that it returns a value. ❑

Quick Quiz 7.25:
How can you tell if one counter is greater than another, while accounting for counter wrap? ■

Answer:
In the C language, the following macro correctly handles this:

```
#define ULONG_CMP_LT(a, b) \
        (ULONG_MAX / 2 < (a) - (b))
```

Although it is tempting to simply subtract two signed integers, this should be avoided because signed overflow is undefined in the C language. For example, if the compiler knows that one of the values is positive and the other negative, it is within its rights to simply assume that the positive number is greater than the negative number, even though subtracting the negative number from the positive number might well result in overflow and thus a negative number.

How could the compiler know the signs of the two numbers? It might be able to deduce it based on prior assignments and comparisons. In this case, if the per-CPU counters were signed, the compiler could deduce that they were always increasing in value, and then might assume that they would never go negative. This assumption could well lead the compiler to generate unfortunate code [McK12d, Reg10]. ❑

Quick Quiz 7.26:
Which is better, the counter approach or the flag approach? ■

Answer:
The flag approach will normally suffer fewer cache misses, but a better answer is to try both and see which works best for your particular workload. ❑

Quick Quiz 7.27:
How can relying on implicit existence guarantees result in a bug? ■

Answer:
Here are some bugs resulting from improper use of implicit existence guarantees:

1. A program writes the address of a global variable to a file, then a later instance of that same program reads that address and attempts to dereference it. This can fail due to address-space randomization, to say nothing of recompilation of the program.

2. A module can record the address of one of its variables in a pointer located in some other module, then attempt to dereference that pointer after the module has been unloaded.

3. A function can record the address of one of its on-stack variables into a global pointer, which some other function might attempt to dereference after that function has returned.

I am sure that you can come up with additional possibilities. ❑

Quick Quiz 7.28:
What if the element we need to delete is not the first element of the list on line 8 of Listing 7.9? ■

Answer:
This is a very simple hash table with no chaining, so the only element in a given bucket is the first element. The reader is invited to adapt this example to a hash table with full chaining. ❑

E.8 Data Ownership

Quick Quiz 8.1:
What form of data ownership is extremely difficult to avoid when creating shared-memory parallel programs (for example, using pthreads) in C or C++? ■

Answer:
Use of auto variables in functions. By default, these are private to the thread executing the current function. ❑

Quick Quiz 8.2:
What synchronization remains in the example shown in Section 8.1? ■

Answer:
The creation of the threads via the `sh &` operator and the joining of thread via the `sh wait` command.

Of course, if the processes explicitly share memory, for example, using the `shmget()` or `mmap()` system calls, explicit synchronization might well be needed when accessing or updating the shared memory. The processes might also synchronize using any of the following interprocess communications mechanisms:

1. System V semaphores.
2. System V message queues.
3. UNIX-domain sockets.
4. Networking protocols, including TCP/IP, UDP, and a whole host of others.
5. File locking.
6. Use of the `open()` system call with the `O_CREAT` and `O_EXCL` flags.
7. Use of the `rename()` system call.

A complete list of possible synchronization mechanisms is left as an exercise to the reader, who is warned that it will be an extremely long list. A surprising number of unassuming system calls can be pressed into service as synchronization mechanisms. ❑

Quick Quiz 8.3:
Is there any shared data in the example shown in Section 8.1? ■

Answer:
That is a philosophical question.

Those wishing the answer "no" might argue that processes by definition do not share memory.

Those wishing to answer "yes" might list a large number of synchronization mechanisms that do not require shared memory, note that the kernel will have some shared state, and perhaps even argue that the assignment of process IDs (PIDs) constitute shared data.

Such arguments are excellent intellectual exercise, and are also a wonderful way of feeling intelligent and scoring points against hapless classmates or colleagues, but are mostly a way of avoiding getting anything useful done. ❑

Quick Quiz 8.4:
Does it ever make sense to have partial data ownership where each thread reads only its own instance of a per-thread variable, but writes to other threads' instances? ■

Answer:
Amazingly enough, yes. One example is a simple message-passing system where threads post messages to other threads' mailboxes, and where each thread is responsible for removing any message it sent once that message has been acted on. Implementation of such an algorithm is left as an exercise for the reader, as is identifying other algorithms with similar ownership patterns. ❑

Quick Quiz 8.5:
What mechanisms other than POSIX signals may be used for function shipping? ■

Answer:
There is a very large number of such mechanisms, including:

1. System V message queues.

www.dbooks.org

2. Shared-memory dequeue (see Section 6.1.2).
3. Shared-memory mailboxes.
4. UNIX-domain sockets.
5. TCP/IP or UDP, possibly augmented by any number of higher-level protocols, including RPC, HTTP, XML, SOAP, and so on.

Compilation of a complete list is left as an exercise to sufficiently single-minded readers, who are warned that the list will be extremely long. ❑

Quick Quiz 8.6:
But none of the data in the `eventual()` function shown on lines 17–34 of Listing 5.5 is actually owned by the `eventual()` thread! In just what way is this data ownership??? ■

Answer:
The key phrase is "owns the rights to the data". In this case, the rights in question are the rights to access the per-thread `counter` variable defined on line 1 of the listing. This situation is similar to that described in Section 8.2.

However, there really is data that is owned by the `eventual()` thread, namely the `t` and `sum` variables defined on lines 19 and 20 of the listing.

For other examples of designated threads, look at the kernel threads in the Linux kernel, for example, those created by `kthread_create()` and `kthread_run()`. ❑

Quick Quiz 8.7:
Is it possible to obtain greater accuracy while still maintaining full privacy of the per-thread data? ■

Answer:
Yes. One approach is for `read_count()` to add the value of its own per-thread variable. This maintains full ownership and performance, but only a slight improvement in accuracy, particularly on systems with very large numbers of threads.

Another approach is for `read_count()` to use function shipping, for example, in the form of per-thread signals. This greatly improves accuracy, but at a significant performance cost for `read_count()`.

However, both of these methods have the advantage of eliminating cache thrashing for the common case of updating counters. ❑

E.9 Deferred Processing

Quick Quiz 9.1:
Why bother with a use-after-free check? ■

Answer:
To greatly increase the probability of finding bugs. A small torture-test program (`routetorture.h`) that allocates and frees only one type of structure can tolerate a surprisingly large amount of use-after-free misbehavior. See Figure 11.4 on page 207 and the related discussion in Section 11.6.4 starting on page 208 for more on the importance of increasing the probability of finding bugs. ❑

Quick Quiz 9.2:
Why doesn't `route_del()` in Listing 9.3 use reference counts to protect the traversal to the element to be freed? ■

Answer:
Because the traversal is already protected by the lock, so no additional protection is required. ❑

Quick Quiz 9.3:
Why the break in the "ideal" line at 224 CPUs in Figure 9.2? Shouldn't it be a straight line? ■

Answer:
The break is due to hyperthreading. On this particular system, the first hardware thread in each core within a socket have consecutive CPU numbers, followed by the first hardware threads in each core for the other sockets, and finally followed by the second hardware thread in each core on all the sockets. On this particular system, CPU numbers 0–27 are the first hardware threads in each of the 28 cores in the first socket, numbers 28–55 are the first hardware threads in each of the 28 cores in the second socket, and so on, so that numbers 196–223 are the first hardware threads in each of the 28 cores in the eighth socket. Then CPU numbers 224–251 are the second hardware threads in each of the 28 cores of the first socket, numbers 252–279 are the second hardware threads in each of the 28 cores of the second socket, and so on until numbers 420–447 are the second hardware threads in each of the 28 cores of the eighth socket.

Why does this matter?

Because the two hardware threads of a given core share resources, and this workload seems to allow a single hardware thread to consume more than half of the relevant

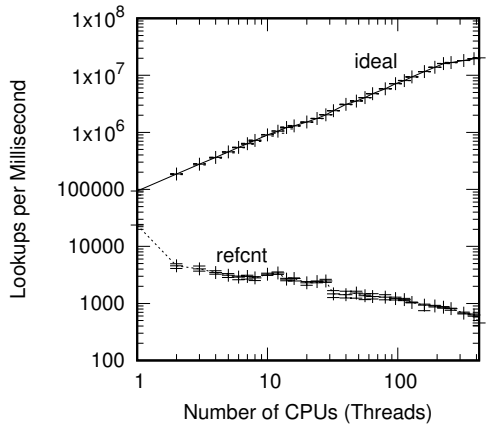


Figure E.4: Pre-BSD Routing Table Protected by Reference Counting, Log Scale

resources within its core. Therefore, adding the second hardware thread of that core adds less than one might hope. Other workloads might gain greater benefit from each core's second hardware thread, but much depends on the details of both the hardware and the workload. ❑

Quick Quiz 9.4:
Shouldn't the refcnt trace in Figure 9.2 be at least a little bit off of the x-axis??? ■

Answer:
Define "a little bit."

Figure E.4 shows the same data, but on a log-log plot. As you can see, the refcnt line drops below 5,000 at two CPUs. This means that the refcnt performance at two CPUs is more than one thousand times smaller than the first y-axis tick of 5×10^6 in Figure 9.2. Therefore, the depiction of the performance of reference counting shown in Figure 9.2 is all too accurate. ❑

Quick Quiz 9.5:
If concurrency has "most definitely reduced the usefulness of reference counting", why are there so many reference counters in the Linux kernel? ■

Answer:
That sentence did say "reduced the usefulness", not "eliminated the usefulness", now didn't it?

Please see Section 13.2, which discusses some of the techniques that the Linux kernel uses to take advantage of reference counting in a highly concurrent environment. ❑

Quick Quiz 9.6:
Given that papers on hazard pointers use the bottom bits of each pointer to mark deleted elements, what is up with `HAZPTR_POISON`? ■

Answer:
The published implementations of hazard pointers used non-blocking synchronization techniques for insertion and deletion. These techniques require that readers traversing the data structure "help" updaters complete their updates, which in turn means that readers need to look at the successor of a deleted element.

In contrast, we will be using locking to synchronize updates, which does away with the need for readers to help updaters complete their updates, which in turn allows us to leave pointers' bottom bits alone. This approach allows read-side code to be simpler and faster. ❑

Quick Quiz 9.7:
Why does `hp_try_record()` in Listing 9.4 take a double indirection to the data element? Why not `void *` instead of `void **`? ■

Answer:
Because `hp_try_record()` must check for concurrent modifications. To do that job, it needs a pointer to a pointer to the element, so that it can check for a modification to the pointer to the element. ❑

Quick Quiz 9.8:
Why bother with `hp_try_record()`? Wouldn't it be easier to just use the failure-immune `hp_record()` function? ■

Answer:
It might be easier in some sense, but as will be seen in the Pre-BSD routing example, there are situations for which `hp_record()` simply does not work. ❑

Quick Quiz 9.9:
Readers must "typically" restart? What are some exceptions? ■

Answer:
If the pointer emanates from a global variable or is otherwise not subject to being freed, then `hp_record()` may be used to repeatedly attempt to record the hazard pointer, even in the face of concurrent deletions.

In certain cases, restart can be avoided by using link counting as exemplified by the UnboundedQueue and

www.dbooks.org

ConcurrentHashMap data structures implemented in Folly open-source library.[9] ❑

Quick Quiz 9.10:
But don't these restrictions on hazard pointers also apply to other forms of reference counting? ■

Answer:
Yes and no. These restrictions apply only to reference-counting mechanisms whose reference acquisition can fail. ❑

Quick Quiz 9.11:
Figure 9.3 shows no sign of hyperthread-induced flattening at 224 threads. Why is that? ■

Answer:
Modern microprocessors are complicated beasts, so significant skepticism is appropriate for any simple answer. That aside, the most likely reason is the full memory barriers required by hazard-pointers readers. Any delays resulting from those memory barriers would make time available to the other hardware thread sharing the core, resulting in greater scalability at the expense of per-hardware-thread performance. ❑

Quick Quiz 9.12:
The paper "Structured Deferral: Synchronization via Procrastination" [McK13] shows that hazard pointers have near-ideal performance. Whatever happened in Figure 9.3??? ■

Answer:
First, Figure 9.3 has a linear y-axis, while most of the graphs in the "Structured Deferral" paper have logscale y-axes. Next, that paper uses lightly-loaded hash tables, while Figure 9.3's uses a 10-element simple linked list, which means that hazard pointers face a larger memory-barrier penalty in this workload than in that of the "Structured Deferral" paper. Finally, that paper used an older modest-sized x86 system, while a much newer and larger system was used to generate the data shown in Figure 9.3.

In addition, use of pairwise asymmetric barriers [Mic08, Cor10b, Cor18] has been proposed to eliminate the read-side hazard-pointer memory barriers on systems supporting this notion [Gol18b], which might improve the performance of hazard pointers beyond what is shown in the figure.

[9] `https://github.com/facebook/folly`

As always, your mileage may vary. Given the difference in performance, it is clear that hazard pointers give you the best performance either for very large data structures (where the memory-barrier overhead will at least partially overlap cache-miss penalties) and for data structures such as hash tables where a lookup operation needs a minimal number of hazard pointers. ❑

Quick Quiz 9.13:
Why isn't this sequence-lock discussion in Chapter 7, you know, the one on *locking*? ■

Answer:
The sequence-lock mechanism is really a combination of two separate synchronization mechanisms, sequence counts and locking. In fact, the sequence-count mechanism is available separately in the Linux kernel via the `write_seqcount_begin()` and `write_seqcount_end()` primitives.

However, the combined `write_seqlock()` and `write_sequnlock()` primitives are used much more heavily in the Linux kernel. More importantly, many more people will understand what you mean if you say "sequence lock" than if you say "sequence count".

So this section is entitled "Sequence Locks" so that people will understand what it is about just from the title, and it appears in the "Deferred Processing" because (1) of the emphasis on the "sequence count" aspect of "sequence locks" and (2) because a "sequence lock" is much more than merely a lock. ❑

Quick Quiz 9.14:
Why not have `read_seqbegin()` in Listing 9.10 check for the low-order bit being set, and retry internally, rather than allowing a doomed read to start? ■

Answer:
That would be a legitimate implementation. However, if the workload is read-mostly, it would likely increase the overhead of the common-case successful read, which could be counter-productive. However, given a sufficiently large fraction of updates and sufficiently high-overhead readers, having the check internal to `read_seqbegin()` might be preferable. ❑

Quick Quiz 9.15:
Why is the `smp_mb()` on line 26 of Listing 9.10 needed? ■

Answer:
If it was omitted, both the compiler and the CPU would be within their rights to move the critical section preceding the call to `read_seqretry()` down below this function. This would prevent the sequence lock from protecting the critical section. The `smp_mb()` primitive prevents such reordering. ❑

Quick Quiz 9.16:
Can't weaker memory barriers be used in the code in Listing 9.10? ■

Answer:
In older versions of the Linux kernel, no.

In very new versions of the Linux kernel, line 16 could use `smp_load_acquire()` instead of `READ_ONCE()`, which in turn would allow the `smp_mb()` on line 17 to be dropped. Similarly, line 41 could use an `smp_store_release()`, for example, as follows:

```
smp_store_release(&slp->seq, READ_ONCE(slp->seq) + 1);
```

This would allow the `smp_mb()` on line 40 to be dropped. ❑

Quick Quiz 9.17:
What prevents sequence-locking updaters from starving readers? ■

Answer:
Nothing. This is one of the weaknesses of sequence locking, and as a result, you should use sequence locking only in read-mostly situations. Unless of course read-side starvation is acceptable in your situation, in which case, go wild with the sequence-locking updates! ❑

Quick Quiz 9.18:
What if something else serializes writers, so that the lock is not needed? ■

Answer:
In this case, the `->lock` field could be omitted, as it is in `seqcount_t` in the Linux kernel. ❑

Quick Quiz 9.19:
Why isn't `seq` on line 2 of Listing 9.10 `unsigned` rather than `unsigned long`? After all, if `unsigned` is good enough for the Linux kernel, shouldn't it be good enough for everyone? ■

Answer:
Not at all. The Linux kernel has a number of special attributes that allow it to ignore the following sequence of events:

1. Thread 0 executes `read_seqbegin()`, picking up `->seq` in line 16, noting that the value is even, and thus returning to the caller.
2. Thread 0 starts executing its read-side critical section, but is then preempted for a long time.
3. Other threads repeatedly invoke `write_seqlock()` and `write_sequnlock()`, until the value of `->seq` overflows back to the value that Thread 0 fetched.
4. Thread 0 resumes execution, completing its read-side critical section with inconsistent data.
5. Thread 0 invokes `read_seqretry()`, which incorrectly concludes that Thread 0 has seen a consistent view of the data protected by the sequence lock.

The Linux kernel uses sequence locking for things that are updated rarely, with time-of-day information being a case in point. This information is updated at most once per millisecond, so that seven weeks would be required to overflow the counter. If a kernel thread was preempted for seven weeks, the Linux kernel's soft-lockup code would be emitting warnings every two minutes for that entire time.

In contrast, with a 64-bit counter, more than five centuries would be required to overflow, even given an update every *nano*second. Therefore, this implementation uses a type for `->seq` that is 64 bits on 64-bit systems. ❑

Quick Quiz 9.20:
Can this bug be fixed? In other words, can you use sequence locks as the *only* synchronization mechanism protecting a linked list supporting concurrent addition, deletion, and lookup? ■

Answer:
One trivial way of accomplishing this is to surround all accesses, including the read-only accesses, with `write_seqlock()` and `write_sequnlock()`. Of course, this

www.dbooks.org

solution also prohibits all read-side parallelism, resulting in massive lock contention, and furthermore could just as easily be implemented using simple locking.

If you do come up with a solution that uses `read_seqbegin()` and `read_seqretry()` to protect read-side accesses, make sure that you correctly handle the following sequence of events:

1. CPU 0 is traversing the linked list, and picks up a pointer to list element A.
2. CPU 1 removes element A from the list and frees it.
3. CPU 2 allocates an unrelated data structure, and gets the memory formerly occupied by element A. In this unrelated data structure, the memory previously used for element A's `->next` pointer is now occupied by a floating-point number.
4. CPU 0 picks up what used to be element A's `->next` pointer, gets random bits, and therefore gets a segmentation fault.

One way to protect against this sort of problem requires use of "type-safe memory", which will be discussed in Section 9.5.4.8. Roughly similar solutions are possible using the hazard pointers discussed in Section 9.3. But in either case, you would be using some other synchronization mechanism in addition to sequence locks! ❑

Quick Quiz 9.21:
Why does Figure 9.7 use `smp_store_release()` given that it is storing a `NULL` pointer? Wouldn't `WRITE_ONCE()` work just as well in this case, given that there is no structure initialization to order against the store of the `NULL` pointer? ■

Answer:
Yes, it would.

Because a `NULL` pointer is being assigned, there is nothing to order against, so there is no need for `smp_store_release()`. In contrast, when assigning a non-`NULL` pointer, it is necessary to use `smp_store_release()` in order to ensure that initialization of the pointed-to structure is carried out before assignment of the pointer.

In short, `WRITE_ONCE()` would work, and would save a little bit of CPU time on some architectures. However, as we will see, software-engineering concerns will motivate use of a special `rcu_assign_pointer()` that is quite similar to `smp_store_release()`. ❑

Quick Quiz 9.22:
Readers running concurrently each other and with the procedure outlined in Figure 9.7 can disagree on the value of `gptr`. Isn't that just a wee bit problematic??? ■

Answer:
Not necessarily.

As hinted at in Sections 3.2.3 and 3.3, speed-of-light delays mean that a computer's data is always stale compared to whatever external reality that data is intended to model.

Real-world algorithms therefore absolutely must tolerate inconsistancies between external reality and the in-computer data reflecting that reality. Many of those algorithms are also able to tolerate some degree of inconsistency within the in-computer data. Section 10.3.3 discusses this point in more detail.

Please note that this need to tolerate inconsistent and stale data is not limited to RCU. It also applies to reference counting, hazard pointers, sequence locks, and even to some locking use cases. For example, if you compute some quantity while holding a lock, but use that quantity after releasing that lock, you might well be using stale data. After all, the data that quantity is based on might change arbitrarily as soon as the lock is released.

So yes, RCU readers can see stale and inconsistent data, but no, this is not necessarily problematic. And, when needed, there are RCU usage patterns that avoid both staleness and inconsistency [ACMS03]. ❑

Quick Quiz 9.23:
In Figure 9.8, the last of CPU 3's readers that could possibly have access to the old data item ended before the grace period even started! So why would any anyone bother waiting until CPU 3's later context switch??? ■

Answer:
Because that waiting is exactly what enables readers to use the same sequence of instructions that is appropriate for single-theaded situations. In other words, this additional "redundant" waiting enables excellent read-side performance, scalability, and real-time response. ❑

Quick Quiz 9.24:
What is the point of `rcu_read_lock()` and `rcu_read_unlock()` in Listing 9.13? Why not just let the quiescent states speak for themselves? ■

Answer:
Recall that readers are not permitted to pass through a quiescent state. For example, within the Linux kernel, RCU readers are not permitted to execute a context switch. Use of `rcu_read_lock()` and `rcu_read_unlock()` enables debug checks for improperly placed quiescent states, making it easy to find bugs that would otherwise be difficult to find, intermittent, and quite destructive. ❑

Quick Quiz 9.25:
What is the point of `rcu_dereference()`, `rcu_assign_pointer()` and `RCU_INIT_POINTER()` in Listing 9.13? Why not just use `READ_ONCE()`, `smp_store_release()`, and `WRITE_ONCE()`, respectively? ■

Answer:
The RCU-specific APIs do have similar semantics to the suggested replacements, but also enable static-analysis debugging checks that complain if an RCU-specific API is invoked on a non-RCU pointer and vice versa. ❑

Quick Quiz 9.26:
But what if the old structure needs to be freed, but the caller of `ins_route()` cannot block, perhaps due to performance considerations or perhaps because the caller is executing within an RCU read-side critical section? ■

Answer:
A `call_rcu()` function, which is described in Section 9.5.2.2, permits asynchronous grace-period waits. ❑

Quick Quiz 9.27:
Doesn't Section 9.4's seqlock also permit readers and updaters to make useful concurrent forward progress? ■

Answer:
Yes and no. Although seqlock readers can run concurrently with seqlock writers, whenever this happens, the `read_seqretry()` primitive will force the reader to retry. This means that any work done by a seqlock reader running concurrently with a seqlock updater will be discarded and the redone upon retry. So seqlock readers can *run* concurrently with updaters, but they cannot actually get any work done in this case.

In contrast, RCU readers can perform useful work even in presence of concurrent RCU updaters.

However, both reference counters (Section 9.2) and hazard pointers (Section 9.3) really do permit useful concurrent forward progress for both updaters and readers, just at somewhat greater cost. Please see Section 9.6 for a comparison of these different solutions to the deferred-reclamation problem. ❑

Quick Quiz 9.28:
Wouldn't use of data ownership for RCU updaters mean that the updates could use exactly the same sequence of instructions as would the corresponding single-threaded code? ■

Answer:
Sometimes, for example, on TSO systems such as x86 or the IBM mainframe where a store-release operation emits a single store instruction. However, weakly ordered systems must also emit a memory barrier or perhaps a store-release instruction. In addition, removing data requires quite a bit of additional work because it is necessary to wait for pre-existing readers before freeing the removed data. ❑

Quick Quiz 9.29:
But suppose that updaters are adding and removing multiple data items from a linked list while a reader is iterating over that same list. Specifically, suppose that a list initially contains elements A, B, and C, and that an updater removes element A and then adds a new element D at the end of the list. The reader might well see {A, B, C, D}, when that sequence of elements never actually ever existed! In what alternate universe would that qualify as "not disrupting concurrent readers"??? ■

Answer:
In the universe where an iterating reader is only required to traverse elements that were present throughout the full duration of the iteration. In the example, that would be elements B and C. Because elements A and D were each present for only part of the iteration, the reader is permitted to iterate over them, but not obliged to. Note that this supports the common case where the reader is simply looking up a single item, and does not know or care about the presence or absence of other items.

If stronger consistency is required, then higher-cost synchronization mechanisms are required, for example, sequence locking or reader-writer locking. But if stronger consistency is *not* required (and it very often is not), then why pay the higher cost? ❑

www.dbooks.org

Quick Quiz 9.30:
What other final values of r1 and r2 are possible in Figure 9.11? ■

Answer:
The r1 == 0 && r2 == 0 possibility was called out in the text. Given that r1 == 0 implies r2 == 0, we know that r1 == 0 && r2 == 1 is forbidden. The following discussion will show that both r1 == 1 && r2 == 1 and r1 == 1 && r2 == 0 are possible. ❑

Quick Quiz 9.31:
What would happen if the order of P0()'s two accesses was reversed in Figure 9.12? ■

Answer:
Absolutely nothing would change. The fact that P0()'s loads from x and y are in the same RCU read-side critical section suffices; their order is irrelevant. ❑

Quick Quiz 9.32:
What would happen if P0()'s accesses in Figures 9.11–9.13 were stores? ■

Answer:
The exact same ordering rules would apply, that is, (1) If any part of P0()'s RCU read-side critical section preceded the beginning of P1()'s grace period, all of P0()'s RCU read-side critical section would precede the end of P1()'s grace period, and (2) If any part of P0()'s RCU read-side critical section followed the end of P1()'s grace period, all of P0()'s RCU read-side critical section would follow the beginning of P1()'s grace period.

It might seem strange to have RCU read-side critical sections containing writes, but RCU is just fine with this. This capability is used frequently in the Linux kernel, for example, acquiring a lock on or reference to a data structure. Acquiring either a lock or a reference results in a write to memory, and it is OK to do these within an RCU read-side critical section.

If having writes in RCU read-side critical sections still seems strange, please review Section 5.4.6, which presented a use case for writes in reader-writer locking read-side critical sections. ❑

Quick Quiz 9.33:
How would you modify the deletion example to permit more than two versions of the list to be active? ■

Answer:
One way of accomplishing this is as shown in Listing E.2.

Listing E.2: Concurrent RCU Deletion

```
1  spin_lock(&mylock);
2  p = search(head, key);
3  if (p == NULL)
4    spin_unlock(&mylock);
5  else {
6    list_del_rcu(&p->list);
7    spin_unlock(&mylock);
8    synchronize_rcu();
9    kfree(p);
10 }
```

Note that this means that multiple concurrent deletions might be waiting in synchronize_rcu(). ❑

Quick Quiz 9.34:
How many RCU versions of a given list can be active at any given time? ■

Answer:
That depends on the synchronization design. If a semaphore protecting the update is held across the grace period, then there can be at most two versions, the old and the new.

However, suppose that only the search, the update, and the list_replace_rcu() were protected by a lock, so that the synchronize_rcu() was outside of that lock, similar to the code shown in Listing E.2. Suppose further that a large number of threads undertook an RCU replacement at about the same time, and that readers are also constantly traversing the data structure.

Then the following sequence of events could occur, starting from the end state of Figure 9.15:

1. Thread A traverses the list, obtaining a reference to Element C.
2. Thread B replaces Element C with a new Element F, then waits for its synchronize_rcu() call to return.
3. Thread C traverses the list, obtaining a reference to Element F.
4. Thread D replaces Element F with a new Element G, then waits for its synchronize_rcu() call to return.
5. Thread E traverses the list, obtaining a reference to Element G.
6. Thread F replaces Element G with a new Element H, then waits for its synchronize_rcu() call to return.
7. Thread G traverses the list, obtaining a reference to Element H.

8. And the previous two steps repeat quickly with additional new elements, so that all of them happen before any of the `synchronize_rcu()` calls return.

Thus, there can be an arbitrary number of versions active, limited only by memory and by how many updates could be completed within a grace period. But please note that data structures that are updated so frequently are not likely to be good candidates for RCU. Nevertheless, RCU can handle high update rates when necessary. ❑

Quick Quiz 9.35:
How can RCU updaters possibly delay RCU readers, given that neither `rcu_read_lock()` nor `rcu_read_unlock()` spin or block? ■

Answer:
The modifications undertaken by a given RCU updater will cause the corresponding CPU to invalidate cache lines containing the data, forcing the CPUs running concurrent RCU readers to incur expensive cache misses. (Can you design an algorithm that changes a data structure *without* inflicting expensive cache misses on concurrent readers? On subsequent readers?) ❑

Quick Quiz 9.36:
Why do some of the cells in Table 9.1 have exclamation marks ("!")? ■

Answer:
The API members with exclamation marks (`rcu_read_lock()`, `rcu_read_unlock()`, and `call_rcu()`) were the only members of the Linux RCU API that Paul E. McKenney was aware of back in the mid-90s. During this timeframe, he was under the mistaken impression that he knew all that there is to know about RCU. ❑

Quick Quiz 9.37:
How do you prevent a huge number of RCU read-side critical sections from indefinitely blocking a `synchronize_rcu()` invocation? ■

Answer:
There is no need to do anything to prevent RCU read-side critical sections from indefinitely blocking a `synchronize_rcu()` invocation, because the `synchronize_rcu()` invocation need wait only for *pre-existing* RCU read-side critical sections. So as long as each RCU read-side critical section is of finite duration, RCU grace periods will also remain finite. ❑

Quick Quiz 9.38:
The `synchronize_rcu()` API waits for all pre-existing interrupt handlers to complete, right? ■

Answer:
In v4.20 and later Linux kernels, yes [McK19c, McK19a].

But not in earlier kernels, and especially not when using preemptible RCU! You instead want `synchronize_irq()`. Alternatively, you can place calls to `rcu_read_lock()` and `rcu_read_unlock()` in the specific interrupt handlers that you want `synchronize_rcu()` to wait for. But even then, be careful, as preemptible RCU will not be guaranteed to wait for that portion of the interrupt handler preceding the `rcu_read_lock()` or following the `rcu_read_unlock()`. ❑

Quick Quiz 9.39:
Under what conditions can `synchronize_srcu()` be safely used within an SRCU read-side critical section? ■

Answer:
In principle, you can use either `synchronize_srcu()` or `synchronize_srcu_expedited()` with a given `srcu_struct` within an SRCU read-side critical section that uses some other `srcu_struct`. In practice, however, doing this is almost certainly a bad idea. In particular, the code shown in Listing E.3 could still result in deadlock. ❑

Listing E.3: Multistage SRCU Deadlocks

```
1 idx = srcu_read_lock(&ssa);
2 synchronize_srcu(&ssb);
3 srcu_read_unlock(&ssa, idx);
4
5 /* . . . */
6
7 idx = srcu_read_lock(&ssb);
8 synchronize_srcu(&ssa);
9 srcu_read_unlock(&ssb, idx);
```

Quick Quiz 9.40:
In a kernel built with `CONFIG_PREEMPT_NONE=y`, won't `synchronize_rcu()` wait for all trampolines, given that preemption is disabled and that trampolines never directly or indirectly invoke `schedule()`? ■

Answer:
You are quite right!

In fact, in nonpreemptible kernels, `synchronize_rcu_tasks()` is a wrapper around `synchronize_rcu()`. ❑

www.dbooks.org

Quick Quiz 9.41:
Normally, any pointer subject to `rcu_dereference()` *must* always be updated using one of the pointer-publish functions in Table 9.2, for example, `rcu_assign_pointer()`.
What is an exception to this rule? ■

Answer:
One such exception is when a multi-element linked data structure is initialized as a unit while inaccessible to other CPUs, and then a single `rcu_assign_pointer()` is used to plant a global pointer to this data structure. The initialization-time pointer assignments need not use `rcu_assign_pointer()`, though any such assignments that happen after the structure is globally visible *must* use `rcu_assign_pointer()`.

However, unless this initialization code is on an impressively hot code-path, it is probably wise to use `rcu_assign_pointer()` anyway, even though it is in theory unnecessary. It is all too easy for a "minor" change to invalidate your cherished assumptions about the initialization happening privately. ❑

Quick Quiz 9.42:
Are there any downsides to the fact that these traversal and update primitives can be used with any of the RCU API family members? ■

Answer:
It can sometimes be difficult for automated code checkers such as "sparse" (or indeed for human beings) to work out which type of RCU read-side critical section a given RCU traversal primitive corresponds to. For example, consider the code shown in Listing E.4.

Listing E.4: Diverse RCU Read-Side Nesting

```
1 rcu_read_lock();
2 preempt_disable();
3 p = rcu_dereference(global_pointer);
4
5 /* . . . */
6
7 preempt_enable();
8 rcu_read_unlock();
```

Is the `rcu_dereference()` primitive in a vanilla RCU critical section or an RCU Sched critical section? What would you have to do to figure this out?

But perhaps after the consolidation of the RCU flavors in the v4.20 Linux kernel we no longer need to care! ❑

Quick Quiz 9.43:
But what if an `hlist_nulls` reader gets moved to some other bucket and then back again? ■

Answer:
One way to handle this is to always move nodes to the beginning of the destination bucket, ensuring that when the reader reaches the end of the list having a matching `NULL` pointer, it will have searched the entire list.

Of course, if there are too many move operations in a hash table with many elements per bucket, the reader might never reach the end of a list. One way of avoiding this in the common case is to keep hash tables well-tuned, thus with short lists. One way of detecting the problem and handling it is for the reader to terminate the search after traversing some large number of nodes, acquire the update-side lock, and redo the search, but this might introduce deadlocks. Another way of avoiding the problem entirely is for readers to search within RCU read-side critical sections, and to wait for an RCU grace period between successive updates. An intermediate position might wait for an RCU grace period every N updates, for some suitable value of N. ❑

Quick Quiz 9.44:
Why isn't there a `rcu_read_lock_tasks_held()` for Tasks RCU? ■

Answer:
Because Tasks RCU does not have read-side markers. Instead, Tasks RCU read-side critical sections are bounded by voluntary context switches. ❑

Quick Quiz 9.45:
Wait, what??? How can RCU QSBR possibly be better than ideal? Just what rubbish definition of ideal would fail to be the best of all possible results??? ■

Answer:
This is an excellent question, and the answer is that modern CPUs and compilers are extremely complex. But before getting into that, it is well worth noting that RCU QSBR's performance advantage appears only in the one-hardware-thread-per-core regime. Once the system is fully loaded, RCU QSBR's performance drops back to ideal.

The RCU variant of the `route_lookup()` search loop actually has one more x86 instruction than does the sequential version, namely the `lea` in the sequence `cmp`, `je`, `mov`, `cmp`, `lea`, and `jne`. This extra instruction is due to the `rcu_head` structure at the beginning of the

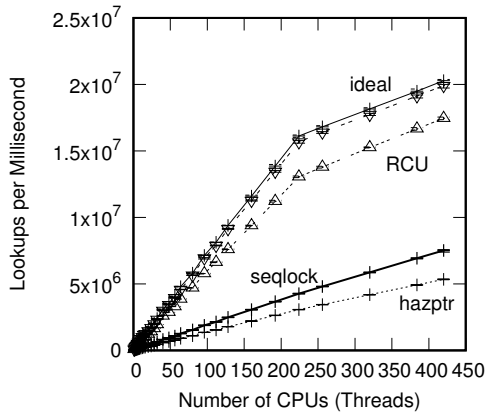


Figure E.5: Pre-BSD Routing Table Protected by RCU QSBR With Non-Initial `rcu_head`

RCU variant's `route_entry` structure, so that, unlike the sequential variant, the RCU variant's `->re_next.next` pointer has a non-zero offset. Back in the 1980s, this additional `lea` instruction might have reliably resulted in the RCU variant being slower, but we are now in the 21st century, and the 1980s are long gone.

But those of you who read Section 3.1.1 carefully already knew all of this!

These counter-intuitive results of course means that any performance result on modern microprocessors must be subject to some skepticism. In theory, it really does not make sense to obtain performance results that are better than ideal, but it really can happen on modern microprocessors. Such results can be thought of as similar to the celebrated super-linear speedups (see Section 6.5 for one such example), that is, of interest but also of limited practical importance. Nevertheless, one of the strengths of RCU is that its read-side overhead is so low that tiny effects such as this one are visible in real performance measurements.

This raises the question as to what would happen if the `rcu_head` structure were to be moved so that RCU's `->re_next.next` pointer also had zero offset, just the same as the sequential variant. And the answer, as can be seen in Figure E.5, is that this causes RCU QSBR's performance to decrease to where it is still very nearly ideal, but no longer super-ideal. ❑

Quick Quiz 9.46:
Given RCU QSBR's read-side performance, why bother with any other flavor of userspace RCU? ■

Answer:
Because RCU QSBR places constraints on the overall application that might not be tolerable, for example, requiring that each and every thread in the application regularly pass through a quiescent state. Among other things, this means that RCU QSBR is not helpful to library writers, who might be better served by other flavors of userspace RCU [MDJ13c]. ❑

Quick Quiz 9.47:
WTF? How the heck do you expect me to believe that RCU can have less than a 300-picosecond overhead when the clock period at 2.10 GHz is almost 500 picoseconds? ■

Answer:
First, consider that the inner loop used to take this measurement is as follows:

```
1  for (i = nloops; i >= 0; i--) {
2    rcu_read_lock();
3    rcu_read_unlock();
4  }
```

Next, consider the effective definitions of `rcu_read_lock()` and `rcu_read_unlock()`:

```
1 #define rcu_read_lock()   barrier()
2 #define rcu_read_unlock() barrier()
```

These definitions constrain compiler code-movement optimizations involving memory references, but emit no instructions in and of themselves. However, if the loop variable is maintained in a register, the accesses to `i` will not count as memory references. Furthermore, the compiler can do loop unrolling, allowing the resulting code to "execute" multiple passes through the loop body simply by incrementing `i` by some value larger than the value 1.

So the "measurement" of 267 picoseconds is simply the fixed overhead of the timing measurements divided by the number of passes through the inner loop containing the calls to `rcu_read_lock()` and `rcu_read_unlock()`, plus the code to manipulate `i` divided by the loop-unrolling factor. And therefore, this measurement really is in error, in fact, it exaggerates the overhead by an arbitrary number of orders of magnitude. After all, in terms of machine instructions emitted, the actual overheads of `rcu_read_lock()` and of `rcu_read_unlock()` are each precisely zero.

It certainly is not just every day that a timing measurement of 267 picoseconds turns out to be an overestimate! ❑

www.dbooks.org

Quick Quiz 9.48:
Didn't an earlier release of this book show RCU read-side overhead way down in the sub-picosecond range? What happened??? ■

Answer:
Excellent memory!!! The overhead in some early releases was in fact roughly 100 femtoseconds.

What happened was that RCU usage spread more broadly through the Linux kernel, including into code that takes page faults. Back at that time, `rcu_read_lock()` and `rcu_read_unlock()` were complete no-ops in `CONFIG_PREEMPT=n` kernels. Unfortunately, that situation allowed the compiler to reorder page-faulting memory accesses into RCU read-side critical sections. Of course, page faults can block, which destroys those critical section.

Nor was this a theoretical problem: A failure actually manifested in 2019. Herbert Xu tracked down this failure down and Linus Torvalds therefore queued a commit to upgrade `rcu_read_lock()` and `rcu_read_unlock()` to unconditionally include a call to `barrier()` [Tor19]. And although `barrier()` emits no code, it does constrain compiler optimizations. And so the price of widespread RCU usage is slightly higher `rcu_read_lock()` and `rcu_read_unlock()` overhead.

Of course, it is also the case that the older results were obtained on a different system than were those shown in Figure 9.23. So which change had the most effect, Linus's commit or the change in the system? This question is left as an exercise to the reader. ❑

Quick Quiz 9.49:
Why is there such large variation for the `rcu` trace in Figure 9.23? ■

Answer:
Keep in mind that this is a log-log plot, so those large-seeming `rcu` variances in reality span only a few hundred picoseconds. And that is such a short time that anything could cause it. However, given that the variance decreases with both small and large numbers of CPUs, one hypothesis is that the variation is due to migrations from one CPU to another.

Yes, these measurements were taken with interrupts disabled, but they were also taken within a guest OS, so that preemption was still possible at the hypervisor level. Attempting to reduce these variations by running the guest OSes at real-time priority (as suggested by Joel Fernandes) is left as an exercise for the reader. ❑

Quick Quiz 9.50:
Given that the system had no fewer than 448 hardware threads, why only 192 CPUs? ■

Answer:
Because the script (`rcuscale.sh`) that generates this data spawn a guest operating system for each set of points gathered, and on this particular system, both `qemu` and KVM limit the number of CPUs that may be configured into a given guest OS. Yes, it would have been possible to run a few more CPUs, but 192 is a nice round number from a binary perspective, given that 256 is infeasible. ❑

Quick Quiz 9.51:
Why the larger error ranges for the submicrosecond durations in Figure 9.25? ■

Answer:
Because smaller disturbances result in greater relative errors for smaller measurements. Also, the Linux kernel's `ndelay()` nanosecond-scale primitive is (as of 2020) less accurate than is the `udelay()` primitive used for the data for durations of a microsecond or more. It is instructive to compare to the zero-length case shown in Figure 9.23. ❑

Quick Quiz 9.52:
Is there an exception to this deadlock immunity, and if so, what sequence of events could lead to deadlock? ■

Answer:
One way to cause a deadlock cycle involving RCU read-side primitives is via the following (illegal) sequence of statements:

```
rcu_read_lock();
synchronize_rcu();
rcu_read_unlock();
```

The `synchronize_rcu()` cannot return until all pre-existing RCU read-side critical sections complete, but is enclosed in an RCU read-side critical section that cannot complete until the `synchronize_rcu()` returns. The result is a classic self-deadlock—you get the same effect when attempting to write-acquire a reader-writer lock while read-holding it.

Note that this self-deadlock scenario does not apply to RCU QSBR, because the context switch performed by the `synchronize_rcu()` would act as a quiescent state for this CPU, allowing a grace period to complete. However, this is if anything even worse, because data used by the

RCU read-side critical section might be freed as a result of the grace period completing.

In short, do not invoke synchronous RCU update-side primitives from within an RCU read-side critical section. ❑

Quick Quiz 9.53:
Immunity to both deadlock and priority inversion??? Sounds too good to be true. Why should I believe that this is even possible? ■

Answer:
It really does work. After all, if it didn't work, the Linux kernel would not run. ❑

Quick Quiz 9.54:
But wait! This is exactly the same code that might be used when thinking of RCU as a replacement for reader-writer locking! What gives? ■

Answer:
This is an effect of the Law of Toy Examples: beyond a certain point, the code fragments look the same. The only difference is in how we think about the code. However, this difference can be extremely important. For but one example of the importance, consider that if we think of RCU as a restricted reference counting scheme, we would never be fooled into thinking that the updates would exclude the RCU read-side critical sections.

It nevertheless is often useful to think of RCU as a replacement for reader-writer locking, for example, when you are replacing reader-writer locking with RCU. ❑

Quick Quiz 9.55:
What if the element we need to delete is not the first element of the list on line 9 of Listing 9.19? ■

Answer:
As with Listing 7.9, this is a very simple hash table with no chaining, so the only element in a given bucket is the first element. The reader is again invited to adapt this example to a hash table with full chaining. Less energetic reader might wish to refer to Chapter 10. ❑

Quick Quiz 9.56:
Why is it OK to exit the RCU read-side critical section on line 15 of Listing 9.19 before releasing the lock on line 17? ■

Answer:
First, please note that the second check on line 14 is necessary because some other CPU might have removed this element while we were waiting to acquire the lock. However, the fact that we were in an RCU read-side critical section while acquiring the lock guarantees that this element could not possibly have been re-allocated and re-inserted into this hash table. Furthermore, once we acquire the lock, the lock itself guarantees the element's existence, so we no longer need to be in an RCU read-side critical section.

The question as to whether it is necessary to re-check the element's key is left as an exercise to the reader. ❑

Quick Quiz 9.57:
Why not exit the RCU read-side critical section on line 23 of Listing 9.19 before releasing the lock on line 22? ■

Answer:
Suppose we reverse the order of these two lines. Then this code is vulnerable to the following sequence of events:

1. CPU 0 invokes `delete()`, and finds the element to be deleted, executing through line 15. It has not yet actually deleted the element, but is about to do so.
2. CPU 1 concurrently invokes `delete()`, attempting to delete this same element. However, CPU 0 still holds the lock, so CPU 1 waits for it at line 13.
3. CPU 0 executes lines 16 and 17, and blocks at line 18 waiting for CPU 1 to exit its RCU read-side critical section.
4. CPU 1 now acquires the lock, but the test on line 14 fails because CPU 0 has already removed the element. CPU 1 now executes line 22 (which we switched with line 23 for the purposes of this Quick Quiz) and exits its RCU read-side critical section.
5. CPU 0 can now return from `synchronize_rcu()`, and thus executes line 19, sending the element to the freelist.
6. CPU 1 now attempts to release a lock for an element that has been freed, and, worse yet, possibly reallocated as some other type of data structure. This is a fatal memory-corruption error. ❑

Quick Quiz 9.58:
But what if there is an arbitrarily long series of RCU

www.dbooks.org

read-side critical sections in multiple threads, so that at any point in time there is at least one thread in the system executing in an RCU read-side critical section? Wouldn't that prevent any data from a SLAB_TYPESAFE_BY_RCU slab ever being returned to the system, possibly resulting in OOM events? ■

Answer:
There could certainly be an arbitrarily long period of time during which at least one thread is always in an RCU read-side critical section. However, the key words in the description in Section 9.5.4.8 are "in-use" and "pre-existing". Keep in mind that a given RCU read-side critical section is conceptually only permitted to gain references to data elements that were in use at the beginning of that critical section. Furthermore, remember that a slab cannot be returned to the system until all of its data elements have been freed, in fact, the RCU grace period cannot start until after they have all been freed.

Therefore, the slab cache need only wait for those RCU read-side critical sections that started before the freeing of the last element of the slab. This in turn means that any RCU grace period that begins after the freeing of the last element will do—the slab may be returned to the system after that grace period ends. ❑

Quick Quiz 9.59:
Suppose that the nmi_profile() function was preemptible. What would need to change to make this example work correctly? ■

Answer:
One approach would be to use rcu_read_lock() and rcu_read_unlock() in nmi_profile(), and to replace the synchronize_sched() with synchronize_rcu(), perhaps as shown in Listing E.5. ❑

Quick Quiz 9.60:
Why not just drop the lock before waiting for the grace period, or using something like call_rcu() instead of waiting for a grace period? ■

Answer:
The authors wished to support linearizable tree operations, so that concurrent additions to, deletions from, and searches of the tree would appear to execute in some globally agreed-upon order. In their search trees, this requires holding locks across grace periods. (It is probably better to drop linearizability as a requirement in most cases, but linearizability is a surprisingly popular (and costly!) requirement.) ❑

Listing E.5: Using RCU to Wait for Mythical Preemptible NMIs to Finish

```
1  struct profile_buffer {
2    long size;
3    atomic_t entry[0];
4  };
5  static struct profile_buffer *buf = NULL;
6
7  void nmi_profile(unsigned long pcvalue)
8  {
9    struct profile_buffer *p;
10
11   rcu_read_lock();
12   p = rcu_dereference(buf);
13   if (p == NULL) {
14     rcu_read_unlock();
15     return;
16   }
17   if (pcvalue >= p->size) {
18     rcu_read_unlock();
19     return;
20   }
21   atomic_inc(&p->entry[pcvalue]);
22   rcu_read_unlock();
23 }
24
25 void nmi_stop(void)
26 {
27   struct profile_buffer *p = buf;
28
29   if (p == NULL)
30     return;
31   rcu_assign_pointer(buf, NULL);
32   synchronize_rcu();
33   kfree(p);
34 }
```

Quick Quiz 9.61:
The statistical-counter implementation shown in Listing 5.4 (count_end.c) used a global lock to guard the summation in read_count(), which resulted in poor performance and negative scalability. How could you use RCU to provide read_count() with excellent performance and good scalability. (Keep in mind that read_count()'s scalability will necessarily be limited by its need to scan all threads' counters.) ■

Answer:
Hint: place the global variable finalcount and the array counterp[] into a single RCU-protected struct. At initialization time, this structure would be allocated and set to all zero and NULL.

The inc_count() function would be unchanged.

The read_count() function would use rcu_read_lock() instead of acquiring final_mutex, and would need to use rcu_dereference() to acquire a reference to the current structure.

The `count_register_thread()` function would set the array element corresponding to the newly created thread to reference that thread's per-thread `counter` variable.

The `count_unregister_thread()` function would need to allocate a new structure, acquire `final_mutex`, copy the old structure to the new one, add the outgoing thread's `counter` variable to the total, `NULL` the pointer to this same `counter` variable, use `rcu_assign_pointer()` to install the new structure in place of the old one, release `final_mutex`, wait for a grace period, and finally free the old structure.

Does this really work? Why or why not?

See Section 13.5.1 on page 265 for more details. ❑

Quick Quiz 9.62:
Section 5.4.6 showed a fanciful pair of code fragments that dealt with counting I/O accesses to removable devices. These code fragments suffered from high overhead on the fastpath (starting an I/O) due to the need to acquire a reader-writer lock. How would you use RCU to provide excellent performance and scalability? (Keep in mind that the performance of the common-case first code fragment that does I/O accesses is much more important than that of the device-removal code fragment.) ■

Answer:
Hint: replace the read-acquisitions of the reader-writer lock with RCU read-side critical sections, then adjust the device-removal code fragment to suit.

See Section 13.5.2 on Page 267 for one solution to this problem. ❑

Quick Quiz 9.63:
Why can't users dynamically allocate the hazard pointers as they are needed? ■

Answer:
They can, but at the expense of additional reader-traversal overhead and, in some environments, the need to handle memory-allocation failure. ❑

Quick Quiz 9.64:
But don't Linux-kernel `kref` reference counters allow guaranteed unconditional reference acquisition? ■

Answer:
Yes they do, but the guarantee only applies unconditionally in cases where a reference is already held. With this in mind, please review the paragraph at the beginning of Section 9.6, especially the part saying "large enough that readers do not hold references from one traversal to another". ❑

Quick Quiz 9.65:
But didn't the answer to one of the quick quizzes in Section 9.3 say that pairwise asymmetric barriers could eliminate the read-side `smp_mb()` from hazard pointers? ■

Answer:
Yes, it did. However, doing this could be argued to change hazard-pointers "Reclamation Forward Progress" row (discussed later) from lock-free to blocking because a CPU spinning with interrupts disabled in the kernel would prevent the update-side portion of the asymmetric barrier from completing. In the Linux kernel, such blocking could in theory be prevented by building the kernel with `CONFIG_NO_HZ_FULL`, designating the relevant CPUs as `nohz_full` at boot time, ensuring that only one thread was ever runnable on a given CPU at a given time, and avoiding ever calling into the kernel. Alternatively, you could ensure that the kernel was free of any bugs that might cause CPUs to spin with interrupts disabled.

Given that CPUs spinning in the Linux kernel with interrupts disabled seems to be rather rare, one might counter-argue that asymmetric-barrier hazard-pointer updates are non-blocking in practice, if not in theory. ❑

E.10 Data Structures

Quick Quiz 10.1:
But chained hash tables are but one type of many. Why the focus on chained hash tables? ■

Answer:
Chained hash tables are completely partitionable, and thus well-suited to concurrent use. There are other completely-partitionable hash tables, for example, split-ordered list [SS06], but they are considerably more complex. We therefore start with chained hash tables. ❑

Quick Quiz 10.2:
But isn't the double comparison on lines 10–13 in Listing 10.3 inefficient in the case where the key fits into an unsigned long? ■

Answer:
Indeed it is! However, hash tables quite frequently store

www.dbooks.org

information with keys such as character strings that do not necessarily fit into an unsigned long. Simplifying the hash-table implementation for the case where keys always fit into unsigned longs is left as an exercise for the reader. ❑

Quick Quiz 10.3:
Instead of simply increasing the number of hash buckets, wouldn't it be better to cache-align the existing hash buckets? ■

Answer:
The answer depends on a great many things. If the hash table has a large number of elements per bucket, it would clearly be better to increase the number of hash buckets. On the other hand, if the hash table is lightly loaded, the answer depends on the hardware, the effectiveness of the hash function, and the workload. Interested readers are encouraged to experiment. ❑

Quick Quiz 10.4:
Given the negative scalability of the Schrödinger's Zoo application across sockets, why not just run multiple copies of the application, with each copy having a subset of the animals and confined to run on a single socket? ■

Answer:
You can do just that! In fact, you can extend this idea to large clustered systems, running one copy of the application on each node of the cluster. This practice is called "sharding", and is heavily used in practice by large web-based retailers [DHJ+07].

However, if you are going to shard on a per-socket basis within a multisocket system, why not buy separate smaller and cheaper single-socket systems, and then run one shard of the database on each of those systems? ❑

Quick Quiz 10.5:
But if elements in a hash table can be removed concurrently with lookups, doesn't that mean that a lookup could return a reference to a data element that was removed immediately after it was looked up? ■

Answer:
Yes it can! This is why `hashtab_lookup()` must be invoked within an RCU read-side critical section, and it is why `hashtab_add()` and `hashtab_del()` must also use RCU-aware list-manipulation primitives. Finally, this is why the caller of `hashtab_del()` must wait for a grace period (e.g., by calling `synchronize_rcu()`) before freeing the removed element. This will ensure that all RCU readers that might reference the newly removed element have completed before that element is freed. ❑

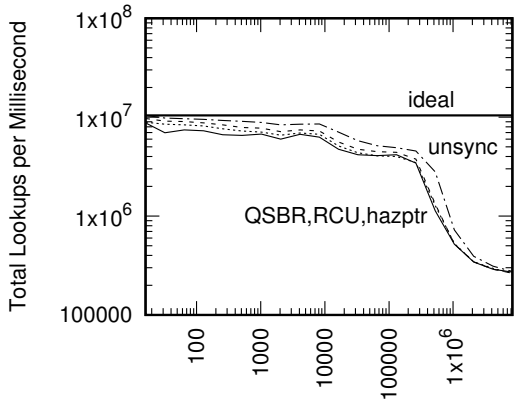


Figure E.6: Read-Only RCU-Protected Hash-Table Performance For Schrödinger's Zoo at 448 CPUs, Varying Table Size

Quick Quiz 10.6:
How can we be so sure that the hash-table size is at fault here, especially given that Figure 10.4 on page 178 shows that varying hash-table size has almost no effect? Might the problem instead be something like false sharing? ■

Answer:
Excellent question!

False sharing requires writes, which are not featured in the unsynchronized and RCU runs of this lookup-only benchmark. The problem is therefore not false sharing.

Still unconvinced? Then look at the log-log plot in Figure E.6, which shows performance for 448 CPUs as a function of the hash-table size, that is, number of buckets and maximum number of elements. A hash-table of size 1,024 has 1,024 buckets and contains at most 1,024 elements, with the average occupancy being 512 elements. Because this is a read-only benchmark, the actual occupancy is always equal to the average occupancy.

This figure shows near-ideal performance below about 8,000 elements, that is, when the hash table comprises less than 1 MB of data. This near-ideal performance is consistent with that for the pre-BSD routing table shown in Figure 9.21 on page 156, even at 448 CPUs. However, the performance drops significantly (this is a log-log plot) at about 8,000 elements, which is where the 1,048,576-byte

L2 cache overflows. Performance falls off a cliff (even on this log-log plot) at about 300,000 elements, where the 40,370,176-byte L3 cache overflows. This demonstrates that the memory-system bottleneck is profound, degrading performance by well in excess of an order of magnitude for the large hash tables. This should not be a surprise, as the size-8,388,608 hash table occupies about 1 GB of memory, overflowing the L3 caches by a factor of 25.

The reason that Figure 10.4 on page 178 shows little effect is that its data was gathered from bucket-locked hash tables, where locking overhead and contention drowned out cache-capacity effects. In contrast, both RCU and hazard-pointers readers avoid stores to shared data, which means that the cache-capacity effects come to the fore.

Still not satisfied? Find a multi-socket system and run this code, making use of whatever performance-counter hardware is available. This hardware should allow you to track down the precise cause of any slowdowns exhibited on your particular system. The experience gained by doing this exercise will be extremely valuable, giving you a significant advantage over those whose understanding of this issue is strictly theoretical.[10] ❑

Quick Quiz 10.7:
The memory system is a serious bottleneck on this big system. Why bother putting 448 CPUs on a system without giving them enough memory bandwidth to do something useful??? ■

Answer:
It would indeed be a bad idea to use this large and expensive system for a workload consisting solely of simple hash-table lookups of small data elements. However, this system is extremely useful for a great many workloads that feature more processing and less memory accessing. For example, some in-memory databases run extremely well on this class of system, albeit when running much more complex sets of queries than performed by the benchmarks in this chapter. For example, such systems might be processing images or video streams stored in each element, providing further performance benefits due to the fact that the resulting sequential memory accesses will make better use of the available memory bandwidth than will a pure pointer-following workload.

But let this be a lesson to you. Modern computer systems come in a great many shapes and sizes, and great care is frequently required to select one that suits your application. And perhaps even more frequently, significant care and work is required to adjust your application to the specific computer systems at hand. ❑

[10] Of course, a theoretical understanding beats no understanding.

Quick Quiz 10.8:
The dangers of extrapolating from 28 CPUs to 448 CPUs was made quite clear in Section 10.2.3. But why should extrapolating up from 448 CPUs be any safer? ■

Answer:
In theory, it isn't any safer, and a useful exercise would be to run these programs on larger systems. In practice, there are a lot more systems with more than 28 CPUs than there are systems with more than 448 CPUs. In addition, other testing has shown that RCU read-side primitives offer consistent performance and scalability up to at least 1024 CPUs. ❑

Quick Quiz 10.9:
How does the code in Listing 10.10 protect against the resizing process progressing past the selected bucket? ■

Answer:
It does not provide any such protection. That is instead the job of the update-side concurrency-control functions described next. ❑

Quick Quiz 10.10:
Suppose that one thread is inserting an element into the hash table during a resize operation. What prevents this insertion from being lost due to a subsequent resize operation completing before the insertion does? ■

Answer:
The second resize operation will not be able to move beyond the bucket into which the insertion is taking place due to the insertion holding the lock(s) on one or both of the hash buckets in the hash tables. Furthermore, the insertion operation takes place within an RCU read-side critical section. As we will see when we examine the `hashtab_resize()` function, this means that each resize operation uses `synchronize_rcu()` invocations to wait for the insertion's read-side critical section to complete. ❑

Quick Quiz 10.11:
The `hashtab_lookup()` function in Listing 10.12 ignores concurrent resize operations. Doesn't this mean that readers might miss an element that was previously added during a resize operation? ■

www.dbooks.org

Answer:
No. As we will see soon, the `hashtab_add()` and `hashtab_del()` functions keep the old hash table up-to-date while a resize operation is in progress. ❑

Quick Quiz 10.12:
The `hashtab_add()` and `hashtab_del()` functions in Listing 10.12 can update two hash buckets while a resize operation is progressing. This might cause poor performance if the frequency of resize operation is not negligible. Isn't it possible to reduce the cost of updates in such cases? ■

Answer:
Yes, at least assuming that a slight increase in the cost of `hashtab_lookup()` is acceptable. One approach is shown in Listings E.6 and E.7 (`hash_resize_s.c`).

Listing E.6: Resizable Hash-Table Access Functions (Fewer Updates)

```
1  struct ht_elem *
2  hashtab_lookup(struct hashtab *htp_master, void *key)
3  {
4    struct ht *htp;
5    struct ht_elem *htep;
6
7    htp = rcu_dereference(htp_master->ht_cur);
8    htep = ht_search_bucket(htp, key);
9    if (htep)
10     return htep;
11   htp = rcu_dereference(htp->ht_new);
12   if (!htp)
13     return NULL;
14   return ht_search_bucket(htp, key);
15 }
16
17 void hashtab_add(struct ht_elem *htep,
18                  struct ht_lock_state *lsp)
19 {
20   struct ht_bucket *htbp = lsp->hbp[0];
21   int i = lsp->hls_idx[0];
22
23   htep->hte_next[!i].prev = NULL;
24   cds_list_add_rcu(&htep->hte_next[i], &htbp->htb_head);
25 }
26
27 void hashtab_del(struct ht_elem *htep,
28                  struct ht_lock_state *lsp)
29 {
30   int i = lsp->hls_idx[0];
31
32   if (htep->hte_next[i].prev) {
33     cds_list_del_rcu(&htep->hte_next[i]);
34     htep->hte_next[i].prev = NULL;
35   }
36   if (lsp->hbp[1] && htep->hte_next[!i].prev) {
37     cds_list_del_rcu(&htep->hte_next[!i]);
38     htep->hte_next[!i].prev = NULL;
39   }
40 }
```

This version of `hashtab_add()` adds an element to either the old bucket if it is not resized yet, or to the new bucket if it has been resized, and `hashtab_del()` removes the specified element from any buckets into which it has been inserted. The `hashtab_lookup()` function searches the new bucket if the search of the old bucket fails, which has the disadvantage of adding overhead to the lookup fastpath. The alternative `hashtab_lock_mod()` returns the locking state of the new bucket in `->hbp[0]` and `->hls_idx[0]` if resize operation is in progress, instead of the perhaps more natural choice of `->hbp[1]` and `->hls_idx[1]`. However, this less-natural choice has the advantage of simplifying `hashtab_add()`.

Further analysis of the code is left as an exercise for the reader. ❑

Listing E.7: Resizable Hash-Table Update-Side Locking Function (Fewer Updates)

```
1  static void
2  hashtab_lock_mod(struct hashtab *htp_master, void *key,
3                   struct ht_lock_state *lsp)
4  {
5    long b;
6    unsigned long h;
7    struct ht *htp;
8    struct ht_bucket *htbp;
9
10   rcu_read_lock();
11   htp = rcu_dereference(htp_master->ht_cur);
12   htbp = ht_get_bucket(htp, key, &b, &h);
13   spin_lock(&htbp->htb_lock);
14   lsp->hbp[0] = htbp;
15   lsp->hls_idx[0] = htp->ht_idx;
16   if (b > READ_ONCE(htp->ht_resize_cur)) {
17     lsp->hbp[1] = NULL;
18     return;
19   }
20   htp = rcu_dereference(htp->ht_new);
21   htbp = ht_get_bucket(htp, key, &b, &h);
22   spin_lock(&htbp->htb_lock);
23   lsp->hbp[1] = lsp->hbp[0];
24   lsp->hls_idx[1] = lsp->hls_idx[0];
25   lsp->hbp[0] = htbp;
26   lsp->hls_idx[0] = htp->ht_idx;
27 }
```

Quick Quiz 10.13:
In the `hashtab_resize()` function in Listing 10.13, what guarantees that the update to `->ht_new` on line 29 will be seen as happening before the update to `->ht_resize_cur` on line 40 from the perspective of `hashtab_add()` and `hashtab_del()`? In other words, what prevents `hashtab_add()` and `hashtab_del()` from dereferencing a NULL pointer loaded from `->ht_new`? ■

Answer:
The `synchronize_rcu()` on line 30 of Listing 10.13 ensures that all pre-existing RCU readers have completed

between the time that we install the new hash-table reference on line 29 and the time that we update `->ht_resize_cur` on line 40. This means that any reader that sees a non-negative value of `->ht_resize_cur` cannot have started before the assignment to `->ht_new`, and thus must be able to see the reference to the new hash table.

And this is why the update-side `hashtab_add()` and `hashtab_del()` functions must be enclosed in RCU read-side critical sections, courtesy of `hashtab_lock_mod()` and `hashtab_unlock_mod()` in Listing 10.11. ❑

Quick Quiz 10.14:
Why is there a `WRITE_ONCE()` on line 40 in Listing 10.13? ■

Answer:
Together with the `READ_ONCE()` on line 16 in `hashtab_lock_mod()` of Listing 10.11, it tells the compiler that the non-initialization accesses to `->ht_resize_cur` must remain because reads from `->ht_resize_cur` really can race with writes, just not in a way to change the "if" conditions. ❑

Quick Quiz 10.15:
How much of the difference in performance between the large and small hash tables shown in Figure 10.19 was due to long hash chains and how much was due to memory-system bottlenecks? ■

Answer:
The easy way to answer this question is to do another run with 2,097,152 elements, but this time also with 2,097,152 buckets, thus bringing the average number of elements per bucket back down to unity.

The results are shown by the triple-dashed new trace in the middle of Figure E.7. The other six traces are identical to their counterparts in Figure 10.19 on page 189. The gap between this new trace and the lower set of three traces is a rough measure of how much of the difference in performance was due to hash-chain length, and the gap between the new trace and the upper set of three traces is a rough measure of how much of that difference was due to memory-system bottlenecks. The new trace starts out slightly below its 262,144-element counterpart at a single CPU, showing that cache capacity is degrading performance slightly even on that single CPU.[11] This is to be expected, given that unlike its smaller counterpart, the

[11] Yes, as far as hardware architects are concerned, caches are part of the memory system.

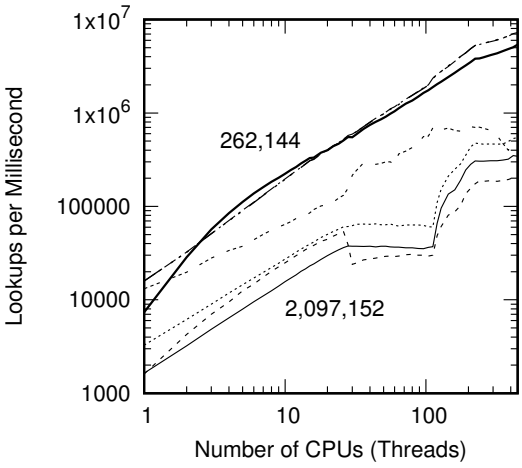


Figure E.7: Effect of Memory-System Bottlenecks on Hash Tables

2,097,152-bucket hash table does not fit into the L3 cache. This new trace rises just past 28 CPUs, which is also to be expected. This rise is due to the fact that the 29th CPU is on another socket, which brings with it an additional 39 MB of cache as well as additional memory bandwidth.

But the large hash table's advantage over that of the hash table with 524,288 buckets (but still 2,097,152 elements) decreases with additional CPUs, which is consistent with the bottleneck residing in the memory system. Above about 400 CPUs, the 2,097,152-bucket hash table is actually outperformed slightly by the 524,288-bucket hash table. This should not be a surprise because the memory system is the bottleneck and the larger number of buckets increases this workload's memory footprint.

The alert reader will have noted the word "roughly" above and might be interested in a more detailed analysis. Such readers are invited to run similar benchmarks, using whatever performance counters or hardware-analysis tools they might have available. This can be a long and complex journey, but those brave enough to embark on it will be rewarded with detailed knowledge of hardware performance and its effect on software. ❑

Quick Quiz 10.16:
Couldn't the `hashtorture.h` code be modified to accommodate a version of `hashtab_lock_mod()` that subsumes the `ht_get_bucket()` functionality? ■

Answer:
It probably could, and doing so would benefit all of the per-bucket-locked hash tables presented in this chapter.

www.dbooks.org

Making this modification is left as an exercise for the reader. ❑

Quick Quiz 10.17:
How much do these specializations really save? Are they really worth it? ■

Answer:
The answer to the first question is left as an exercise to the reader. Try specializing the resizable hash table and see how much performance improvement results. The second question cannot be answered in general, but must instead be answered with respect to a specific use case. Some use cases are extremely sensitive to performance and scalability, while others are less so. ❑

E.11 Validation

Quick Quiz 11.1:
When in computing is it necessary to follow a fragmentary plan? ■

Answer:
There are any number of situations, but perhaps the most important situation is when no one has ever created anything resembling the program to be developed. In this case, the only way to create a credible plan is to implement the program, create the plan, and implement it a second time. But whoever implements the program for the first time has no choice but to follow a fragmentary plan because any detailed plan created in ignorance cannot survive first contact with the real world.

And perhaps this is one reason why evolution has favored insanely optimistic human beings who are happy to follow fragmentary plans! ❑

Quick Quiz 11.2:
Who cares about the organization? After all, it is the project that is important! ■

Answer:
Yes, projects are important, but if you like being paid for your work, you need organizations as well as projects. ❑

Quick Quiz 11.3:
Suppose that you are writing a script that processes the output of the `time` command, which looks as follows:

```
real    0m0.132s
user    0m0.040s
sys     0m0.008s
```

The script is required to check its input for errors, and to give appropriate diagnostics if fed erroneous `time` output. What test inputs should you provide to this program to test it for use with `time` output generated by single-threaded programs? ■

Answer:
Can you say "Yes" to all the following questions?

1. Do you have a test case in which all the time is consumed in user mode by a CPU-bound program?
2. Do you have a test case in which all the time is consumed in system mode by a CPU-bound program?
3. Do you have a test case in which all three times are zero?
4. Do you have a test case in which the "user" and "sys" times sum to more than the "real" time? (This would of course be completely legitimate in a multithreaded program.)
5. Do you have a set of tests cases in which one of the times uses more than one second?
6. Do you have a set of tests cases in which one of the times uses more than ten seconds?
7. Do you have a set of test cases in which one of the times has non-zero minutes? (For example, "15m36.342s".)
8. Do you have a set of test cases in which one of the times has a seconds value of greater than 60?
9. Do you have a set of test cases in which one of the times overflows 32 bits of milliseconds? 64 bits of milliseconds?
10. Do you have a set of test cases in which one of the times is negative?
11. Do you have a set of test cases in which one of the times has a positive minutes value but a negative seconds value?
12. Do you have a set of test cases in which one of the times omits the "m" or the "s"?

13. Do you have a set of test cases in which one of the times is non-numeric? (For example, "Go Fish".)

14. Do you have a set of test cases in which one of the lines is omitted? (For example, where there is a "real" value and a "sys" value, but no "user" value.)

15. Do you have a set of test cases where one of the lines is duplicated? Or duplicated, but with a different time value for the duplicate?

16. Do you have a set of test cases where a given line has more than one time value? (For example, "real 0m0.132s 0m0.008s".)

17. Do you have a set of test cases containing random characters?

18. In all test cases involving invalid input, did you generate all permutations?

19. For each test case, do you have an expected outcome for that test?

If you did not generate test data for a substantial number of the above cases, you will need to cultivate a more destructive attitude in order to have a chance of generating high-quality tests.

Of course, one way to economize on destructiveness is to generate the tests with the to-be-tested source code at hand, which is called white-box testing (as opposed to black-box testing). However, this is no panacea: You will find that it is all too easy to find your thinking limited by what the program can handle, thus failing to generate truly destructive inputs. ❑

Quick Quiz 11.4:
You are asking me to do all this validation BS before I even start coding??? That sounds like a great way to never get started!!! ■

Answer:
If it is your project, for example, a hobby, do what you like. Any time you waste will be your own, and you have no one else to answer to for it. And there is a good chance that the time will not be completely wasted. For example, if you are embarking on a first-of-a-kind project, the requirements are in some sense unknowable anyway. In this case, the best approach might be to quickly prototype a number of rough solutions, try them out, and see what works best.

On the other hand, if you are being paid to produce a system that is broadly similar to existing systems, you owe it to your users, your employer, and your future self to validate early and often. ❑

Quick Quiz 11.5:
Are you actually suggesting that it is possible to test correctness into software??? Everyone knows that is impossible!!! ■

Answer:
Please note that the text used the word "validation" rather than the word "testing". The word "validation" includes formal methods as well as testing, for more on which please see Chapter 12.

But as long as we are bringing up things that everyone should know, let's remind ourselves that Darwinian evolution is not about correctness, but rather about survival. As is software. My goal as a developer is not that my software be attractive from a theoretical viewpoint, but rather that it survive whatever its users throw at it.

Although the notion of correctness does have its uses, its fundamental limitation is that the specification against which correctness is judged will also have bugs. This means nothing more nor less than that traditional correctness proofs prove that the code in question contains the intended set of bugs!

Alternative definitions of correctness instead focus on the lack of problematic properties, for example, proving that the software has no use-after-free bugs, no `NULL` pointer dereferences, no array-out-of-bounds references, and so on. Make no mistake, finding and eliminating such classes of bugs can be highly useful. But the fact remains that the lack of certain classes of bugs does nothing to demonstrate fitness for any specific purpose.

Therefore, usage-driven validation remains critically important.

Besides, it is also impossible to verify correctness into your software, especially given the problematic need to verify both the verifier and the specification. ❑

Quick Quiz 11.6:
How can you implement `WARN_ON_ONCE()`? ■

Answer:
If you don't mind `WARN_ON_ONCE()` sometimes warning more than once, simply maintain a static variable that is initialized to zero. If the condition triggers, check the variable, and if it is non-zero, return. Otherwise, set it to one, print the message, and return.

www.dbooks.org

If you really need the message to never appear more than once, you can use an atomic exchange operation in place of "set it to one" above. Print the message only if the atomic exchange operation returns zero. ❑

Quick Quiz 11.7:
Just what invalid assumptions are you accusing Linux kernel hackers of harboring??? ■

Answer:
Those wishing a complete answer to this question are encouraged to search the Linux kernel *git* repository for commits containing the string "Fixes:". There were many thousands of them just in the year 2020, including fixes for the following invalid assumptions:

1. Testing for a non-zero denominator will prevent divide-by-zero errors. (Hint: Suppose that the test uses 64-bit arithmetic but that the division uses 32-bit arithmetic.)
2. Userspace can be trusted to zero out versioned data structures used to communicate with the kernel. (Hint: Sometimes userspace has no idea how large the data structure is.)
3. Outdated TCP duplicate selective acknowledgement (D-SACK) packets can be completely ignored. (Hint: These packets might also contain other information.)
4. All CPUs are little-endian.
5. Once a data structure is no longer needed, all of its memory may be immediately freed.
6. All devices can be initialized while in standby mode.
7. Developers can be trusted to consistently do correct hexidecimal arithmetic.

Those who look at these commits in greater detail will conclude that invalid assumptions are the rule, not the exception. ❑

Quick Quiz 11.8:
Why would anyone bother copying existing code in pen on paper??? Doesn't that just increase the probability of transcription errors? ■

Answer:
If you are worried about transcription errors, please allow me to be the first to introduce you to a really cool tool named `diff`. In addition, carrying out the copying can be quite valuable:

1. If you are copying a lot of code, you are probably failing to take advantage of an opportunity for abstraction. The act of copying code can provide great motivation for abstraction.
2. Copying the code gives you an opportunity to think about whether the code really works in its new setting. Is there some non-obvious constraint, such as the need to disable interrupts or to hold some lock?
3. Copying the code also gives you time to consider whether there is some better way to get the job done.

So, yes, copy the code! ❑

Quick Quiz 11.9:
This procedure is ridiculously over-engineered! How can you expect to get a reasonable amount of software written doing it this way??? ■

Answer:
Indeed, repeatedly copying code by hand is laborious and slow. However, when combined with heavy-duty stress testing and proofs of correctness, this approach is also extremely effective for complex parallel code where ultimate performance and reliability are required and where debugging is difficult. The Linux-kernel RCU implementation is a case in point.

On the other hand, if you are writing a simple single-threaded shell script, then you would be best-served by a different methodology. For example, enter each command one at a time into an interactive shell with a test data set to make sure that it does what you want, then copy-and-paste the successful commands into your script. Finally, test the script as a whole.

If you have a friend or colleague who is willing to help out, pair programming can work very well, as can any number of formal design- and code-review processes.

And if you are writing code as a hobby, then do whatever you like.

In short, different types of software need different development methodologies. ❑

Quick Quiz 11.10:
What do you do if, after all the pen-on-paper copying, you find a bug while typing in the resulting code? ■

Answer:
The answer, as is often the case, is "it depends". If the bug is a simple typo, fix that typo and continue typing. However, if the bug indicates a design flaw, go back to pen and paper. ❑

Quick Quiz 11.11:
Wait! Why on earth would an abstract piece of software fail only sometimes??? ■

Answer:
Because complexity and concurrency can produce results that are indistinguishable from randomness [MOZ09]. For example, a bug in Linux-kernel RCU required the following to hold before that bug would manifest:

1. The kernel was built for HPC or real-time use, so that a given CPU's RCU work could be offloaded to some other CPU.
2. An offloaded CPU went offline just after generating a large quantity of RCU work.
3. A special `rcu_barrier()` API was invoked just at this time.
4. The RCU work from the newly offlined CPU was still being processed after `rcu_barrier()` returned.
5. One of these remaining RCU work items was related to the code invoking the `rcu_barrier()`.

Making this bug manifest therefore required considerable luck or great testing skill. But the testing skill could be effective only if the bug was known, which of course it was not. Therefore, the manifesting of this bug was very well modeled as a probabilistic process. ❑

Quick Quiz 11.12:
Suppose that you had a very large number of systems at your disposal. For example, at current cloud prices, you can purchase a huge amount of CPU time at low cost. Why not use this approach to get close enough to certainty for all practical purposes? ■

Answer:
This approach might well be a valuable addition to your validation arsenal. But it does have limitations that rule out "for all practical purposes":

1. Some bugs have extremely low probabilities of occurrence, but nevertheless need to be fixed. For example, suppose that the Linux kernel's RCU implementation had a bug that is triggered only once per million years of machine time on average. A million years of CPU time is hugely expensive even on the cheapest cloud platforms, but we could expect this bug to result in more than 50 failures per day on the more than 20 billion Linux instances in the world as of 2017.
2. The bug might well have zero probability of occurrence on your particular cloud-computing test setup, which means that you won't see it no matter how much machine time you burn testing it. For but one example, there are RCU bugs that appear only in preemptible kernels, and also other RCU bugs that appear only in non-preemptible kernels.

Of course, if your code is small enough, formal validation may be helpful, as discussed in Chapter 12. But beware: formal validation of your code will not find errors in your assumptions, misunderstanding of the requirements, misunderstanding of the software or hardware primitives you use, or errors that you did not think to construct a proof for. ❑

Quick Quiz 11.13:
Say what??? When I plug the earlier five-test 10 %-failure-rate example into the formula, I get 59,050 % and that just doesn't make sense!!! ■

Answer:
You are right, that makes no sense at all.

Remember that a probability is a number between zero and one, so that you need to divide a percentage by 100 to get a probability. So 10 % is a probability of 0.1, which gets a probability of 0.4095, which rounds to 41 %, which quite sensibly matches the earlier result. ❑

Quick Quiz 11.14:
In Equation 11.6, are the logarithms base-10, base-2, or base-e? ■

Answer:
It does not matter. You will get the same answer no matter what base of logarithms you use because the result is a pure ratio of logarithms. The only constraint is that you use the same base for both the numerator and the denominator. ❑

Quick Quiz 11.15:
Suppose that a bug causes a test failure three times per hour on average. How long must the test run error-free to provide 99.9 % confidence that the fix significantly reduced the probability of failure? ■

Answer:
We set n to 3 and P to 99.9 in Equation 11.11, resulting in:

$$T = -\frac{1}{3} \ln \frac{100 - 99.9}{100} = 2.3 \tag{E.9}$$

www.dbooks.org

Table E.3: Human-Friendly Poisson-Function Display

	Improvement		
Certainty (%)	Any	10x	100x
90.0	2.3	23.0	230.0
95.0	3.0	30.0	300.0
99.0	4.6	46.1	460.5
99.9	6.9	69.1	690.7

If the test runs without failure for 2.3 hours, we can be 99.9 % certain that the fix reduced the probability of failure. ❑

Quick Quiz 11.16:
Doing the summation of all the factorials and exponentials is a real pain. Isn't there an easier way? ■

Answer:
One approach is to use the open-source symbolic manipulation program named "maxima". Once you have installed this program, which is a part of many Linux distributions, you can run it and give the `load(distrib);` command followed by any number of `bfloat(cdf_poisson(m,l));` commands, where the `m` is replaced by the desired value of m (the actual number of failures in actual test) and the `l` is replaced by the desired value of λ (the expected number of failures in the actual test).

In particular, the `bfloat(cdf_poisson(2,24));` command results in `1.181617112359357b-8`, which matches the value given by Equation 11.13.

Another approach is to recognize that in this real world, it is not all that useful to compute (say) the duration of a test having two or fewer errors that would give a 76.8 % confidence of a 349.2x improvement in reliability. Instead, human beings tend to focus on specific values, for example, a 95 % confidence of a 10x improvement. People also greatly prefer error-free test runs, and so should you because doing so reduces your required test durations. Therefore, it is quite possible that the values in Table E.3 will suffice. Simply look up the desired confidence and degree of improvement, and the resulting number will give you the required error-free test duration in terms of the expected time for a single error to appear. So if your pre-fix testing suffered one failure per hour, and the powers that be require a 95 % confidence of a 10x improvement, you need a 30-hour error-free run.

Alternatively, you can use the rough-and-ready method described in Section 11.6.2. ❑

Quick Quiz 11.17:
But wait!!! Given that there has to be *some* number of failures (including the possibility of zero failures), shouldn't Equation 11.13 approach the value 1 as m goes to infinity? ■

Answer:
Indeed it should. And it does.

To see this, note that $e^{-\lambda}$ does not depend on i, which means that it can be pulled out of the summation as follows:

$$e^{-\lambda} \sum_{i=0}^{\infty} \frac{\lambda^i}{i!} \tag{E.10}$$

The remaining summation is exactly the Taylor series for e^{λ}, yielding:

$$e^{-\lambda} e^{\lambda} \tag{E.11}$$

The two exponentials are reciprocals, and therefore cancel, resulting in exactly 1, as required. ❑

Quick Quiz 11.18:
How is this approach supposed to help if the corruption affected some unrelated pointer, which then caused the corruption??? ■

Answer:
Indeed, that can happen. Many CPUs have hardware-debugging facilities that can help you locate that unrelated pointer. Furthermore, if you have a core dump, you can search the core dump for pointers referencing the corrupted region of memory. You can also look at the data layout of the corruption, and check pointers whose type matches that layout.

You can also step back and test the modules making up your program more intensively, which will likely confine the corruption to the module responsible for it. If this makes the corruption vanish, consider adding additional argument checking to the functions exported from each module.

Nevertheless, this is a hard problem, which is why I used the words "a bit of a dark art". ❑

Quick Quiz 11.19:
But I did the bisection, and ended up with a huge commit. What do I do now? ■

Answer:
A huge commit? Shame on you! This is but one reason why you are supposed to keep the commits small.

And that is your answer: Break up the commit into bite-sized pieces and bisect the pieces. In my experience, the act of breaking up the commit is often sufficient to make the bug painfully obvious. ❑

Quick Quiz 11.20:
Why don't conditional-locking primitives provide this spurious-failure functionality? ■

Answer:
There are locking algorithms that depend on conditional-locking primitives telling them the truth. For example, if conditional-lock failure signals that some other thread is already working on a given job, spurious failure might cause that job to never get done, possibly resulting in a hang. ❑

Quick Quiz 11.21:
That is ridiculous!!! After all, isn't getting the correct answer later than one would like better than getting an incorrect answer??? ■

Answer:
This question fails to consider the option of choosing not to compute the answer at all, and in doing so, also fails to consider the costs of computing the answer. For example, consider short-term weather forecasting, for which accurate models exist, but which require large (and expensive) clustered supercomputers, at least if you want to actually run the model faster than the weather.

And in this case, any performance bug that prevents the model from running faster than the actual weather prevents any forecasting. Given that the whole purpose of purchasing the large clustered supercomputers was to forecast weather, if you cannot run the model faster than the weather, you would be better off not running the model at all.

More severe examples may be found in the area of safety-critical real-time computing. ❑

Quick Quiz 11.22:
But if you are going to put in all the hard work of parallelizing an application, why not do it right? Why settle for anything less than optimal performance and linear scalability? ■

Answer:
Although I do heartily salute your spirit and aspirations, you are forgetting that there may be high costs due to delays in the program's completion. For an extreme example, suppose that a 40 % performance shortfall from a single-threaded application is causing one person to die each day. Suppose further that in a day you could hack together a quick and dirty parallel program that ran 50 % faster on an eight-CPU system than the sequential version, but that an optimal parallel program would require four months of painstaking design, coding, debugging, and tuning.

It is safe to say that more than 100 people would prefer the quick and dirty version. ❑

Quick Quiz 11.23:
But what about other sources of error, for example, due to interactions between caches and memory layout? ■

Answer:
Changes in memory layout can indeed result in unrealistic decreases in execution time. For example, suppose that a given microbenchmark almost always overflows the L0 cache's associativity, but with just the right memory layout, it all fits. If this is a real concern, consider running your microbenchmark using huge pages (or within the kernel or on bare metal) in order to completely control the memory layout.

But note that there are many different possible memory-layout bottlenecks. Benchmarks sensitive to memory bandwidth (such as those involving matrix arithmetic) should spread the running threads across the available cores and sockets to maximize memory parallelism. They should also spread the data across NUMA nodes, memory controllers, and DRAM chips to the extent possible. In contrast, benchmarks sensitive to memory latency (including most poorly scaling applications) should instead maximize locality, filling each core and socket in turn before adding another one. ❑

Quick Quiz 11.24:
Wouldn't the techniques suggested to isolate the code under test also affect that code's performance, particularly if it is running within a larger application? ■

Answer:
Indeed it might, although in most microbenchmarking

www.dbooks.org

efforts you would extract the code under test from the enclosing application. Nevertheless, if for some reason you must keep the code under test within the application, you will very likely need to use the techniques discussed in Section 11.7.6. ❑

Quick Quiz 11.25:
This approach is just plain weird! Why not use means and standard deviations, like we were taught in our statistics classes? ■

Answer:
Because mean and standard deviation were not designed to do this job. To see this, try applying mean and standard deviation to the following data set, given a 1 % relative error in measurement:

49,548.4 49,549.4 49,550.2 49,550.9 49,550.9
49,551.0 49,551.5 49,552.1 49,899.0 49,899.3
49,899.7 49,899.8 49,900.1 49,900.4 52,244.9
53,333.3 53,333.3 53,706.3 53,706.3 54,084.5

The problem is that mean and standard deviation do not rest on any sort of measurement-error assumption, and they will therefore see the difference between the values near 49,500 and those near 49,900 as being statistically significant, when in fact they are well within the bounds of estimated measurement error.

Of course, it is possible to create a script similar to that in Listing 11.2 that uses standard deviation rather than absolute difference to get a similar effect, and this is left as an exercise for the interested reader. Be careful to avoid divide-by-zero errors arising from strings of identical data values! ❑

Quick Quiz 11.26:
But what if all the y-values in the trusted group of data are exactly zero? Won't that cause the script to reject any non-zero value? ■

Answer:
Indeed it will! But if your performance measurements often produce a value of exactly zero, perhaps you need to take a closer look at your performance-measurement code.

Note that many approaches based on mean and standard deviation will have similar problems with this sort of dataset. ❑

E.12 Formal Verification

Quick Quiz 12.1:
Why is there an unreached statement in locker? After all, isn't this a *full* state-space search? ■

Answer:
The locker process is an infinite loop, so control never reaches the end of this process. However, since there are no monotonically increasing variables, Promela is able to model this infinite loop with a small number of states. ❑

Quick Quiz 12.2:
What are some Promela code-style issues with this example? ■

Answer:
There are several:

1. The declaration of `sum` should be moved to within the init block, since it is not used anywhere else.
2. The assertion code should be moved outside of the initialization loop. The initialization loop can then be placed in an atomic block, greatly reducing the state space (by how much?).
3. The atomic block covering the assertion code should be extended to include the initialization of `sum` and `j`, and also to cover the assertion. This also reduces the state space (again, by how much?). ❑

Quick Quiz 12.3:
Is there a more straightforward way to code the `do-od` statement? ■

Answer:
Yes. Replace it with `if-fi` and remove the two `break` statements. ❑

Quick Quiz 12.4:
Why are there atomic blocks at lines 12–21 and lines 44–56, when the operations within those atomic blocks have no atomic implementation on any current production microprocessor? ■

Answer:
Because those operations are for the benefit of the assertion only. They are not part of the algorithm itself. There is therefore no harm in marking them atomic, and so marking them greatly reduces the state space that must be searched by the Promela model. ❑

Quick Quiz 12.5:
Is the re-summing of the counters on lines 24–27 *really* necessary? ■

Answer:
Yes. To see this, delete these lines and run the model.

Alternatively, consider the following sequence of steps:

1. One process is within its RCU read-side critical section, so that the value of `ctr[0]` is zero and the value of `ctr[1]` is two.
2. An updater starts executing, and sees that the sum of the counters is two so that the fastpath cannot be executed. It therefore acquires the lock.
3. A second updater starts executing, and fetches the value of `ctr[0]`, which is zero.
4. The first updater adds one to `ctr[0]`, flips the index (which now becomes zero), then subtracts one from `ctr[1]` (which now becomes one).
5. The second updater fetches the value of `ctr[1]`, which is now one.
6. The second updater now incorrectly concludes that it is safe to proceed on the fastpath, despite the fact that the original reader has not yet completed. ❑

Quick Quiz 12.6:
A compression rate of 0.48 % corresponds to a 200-to-1 decrease in memory occupied by the states! Is the state-space search *really* exhaustive??? ■

Answer:
According to Spin's documentation, yes, it is.

As an indirect evidence, let's compare the results of runs with `-DCOLLAPSE` and with `-DMA=88` (two readers and three updaters). The diff of outputs from those runs is shown in Listing E.8. As you can see, they agree on the numbers of states (stored and matched). ❑

Quick Quiz 12.7:
But different formal-verification tools are often designed to locate particular classes of bugs. For example, very few formal-verification tools will find an error in the specification. So isn't this "clearly untrustworthy" judgment a bit harsh? ■

Listing E.8: Spin Output Diff of `-DCOLLAPSE` and `-DMA=88`

```
@@ -1,6 +1,6 @@
 (Spin Version 6.4.6 -- 2 December 2016)
         + Partial Order Reduction
-        + Compression
+        + Graph Encoding (-DMA=88)

 Full statespace search for:
         never claim             - (none specified)
@@ -9,27 +9,22 @@
         invalid end states      +

 State-vector 88 byte, depth reached 328014, errors: 0
+MA stats: -DMA=77 is sufficient
+Minimized Automaton:    2084798 nodes and 6.38445e+06 edges
 1.8620286e+08 states, stored
 1.7759831e+08 states, matched
 3.6380117e+08 transitions (= stored+matched)
 1.3724093e+08 atomic steps
-hash conflicts: 1.1445626e+08 (resolved)

 Stats on memory usage (in Megabytes):
 20598.919       equivalent memory usage for states
                   (stored*(State-vector + overhead))
- 8418.559       actual memory usage for states
-                  (compression: 40.87%)
-                state-vector as stored =
-                  19 byte + 28 byte overhead
- 2048.000       memory used for hash table (-w28)
+  204.907       actual memory usage for states
+                  (compression: 0.99%)
    17.624       memory used for DFS stack (-m330000)
-    1.509       memory lost to fragmentation
-10482.675       total actual memory usage
+  222.388       total actual memory usage

-nr of templates: [ 0:globals 1:chans 2:procs ]
-collapse counts: [ 0:1021 2:32 3:1869 4:2 ]
 unreached in proctype qrcu_reader
         (0 of 18 states)
 unreached in proctype qrcu_updater
@@ -38,5 +33,5 @@
 unreached in init
         (0 of 23 states)

-pan: elapsed time 369 seconds
-pan: rate 505107.58 states/second
+pan: elapsed time 2.68e+03 seconds
+pan: rate 69453.282 states/second
```


www.dbooks.org

Answer:
It is certainly true that many formal-verification tools are specialized in some way. For example, Promela does not handle realistic memory models (though they can be programmed into Promela [DMD13]), CBMC [CKL04] does not detect probabilistic hangs and deadlocks, and Nidhugg [LSLK14] does not detect bugs involving data nondeterminism. But this means that these tools cannot be trusted to find bugs that they are not designed to locate.

And therefore people creating formal-verification tools should "tell the truth on the label", clearly calling out what classes of bugs their tools can and cannot detect. Otherwise, the first time a practitioner finds a tool failing to detect a bug, that practitioner is likely to make extremely harsh and extremely public denunciations of that tool. Yes, yes, there is something to be said for putting your best foot forward, but putting it too far forward without appropriate disclaimers can easily trigger a land mine of negative reaction that your tool might or might not be able to recover from.

You have been warned! ❑

Quick Quiz 12.8:
Given that we have two independent proofs of correctness for the QRCU algorithm described herein, and given that the proof of incorrectness covers what is known to be a different algorithm, why is there any room for doubt? ■

Answer:
There is always room for doubt. In this case, it is important to keep in mind that the two proofs of correctness preceded the formalization of real-world memory models, raising the possibility that these two proofs are based on incorrect memory-ordering assumptions. Furthermore, since both proofs were constructed by the same person, it is quite possible that they contain a common error. Again, there is always room for doubt. ❑

Quick Quiz 12.9:
Yeah, that's just great! Now, just what am I supposed to do if I don't happen to have a machine with 40 GB of main memory??? ■

Answer:
Relax, there are a number of lawful answers to this question:

1. Try compiler flags `-DCOLLAPSE` and `-DMA=N` to reduce memory consumption. See Section 12.1.4.1.
2. Further optimize the model, reducing its memory consumption.
3. Work out a pencil-and-paper proof, perhaps starting with the comments in the code in the Linux kernel.
4. Devise careful torture tests, which, though they cannot prove the code correct, can find hidden bugs.
5. There is some movement towards tools that do model checking on clusters of smaller machines. However, please note that we have not actually used such tools myself, courtesy of some large machines that Paul has occasional access to.
6. Wait for memory sizes of affordable systems to expand to fit your problem.
7. Use one of a number of cloud-computing services to rent a large system for a short time period. ❑

Quick Quiz 12.10:
Why not simply increment `rcu_update_flag`, and then only increment `dynticks_progress_counter` if the old value of `rcu_update_flag` was zero??? ■

Answer:
This fails in presence of NMIs. To see this, suppose an NMI was received just after `rcu_irq_enter()` incremented `rcu_update_flag`, but before it incremented `dynticks_progress_counter`. The instance of `rcu_irq_enter()` invoked by the NMI would see that the original value of `rcu_update_flag` was non-zero, and would therefore refrain from incrementing `dynticks_progress_counter`. This would leave the RCU grace-period machinery no clue that the NMI handler was executing on this CPU, so that any RCU read-side critical sections in the NMI handler would lose their RCU protection.

The possibility of NMI handlers, which, by definition cannot be masked, does complicate this code. ❑

Quick Quiz 12.11:
But if line 7 finds that we are the outermost interrupt, wouldn't we *always* need to increment `dynticks_progress_counter`? ■

Answer:
Not if we interrupted a running task! In that case, `dynticks_progress_counter` would have already been incremented by `rcu_exit_nohz()`, and there would be no need to increment it again. ❑

Quick Quiz 12.12:
Can you spot any bugs in any of the code in this section? ■

Answer:
Read the next section to see if you were correct. ❑

Quick Quiz 12.13:
Why isn't the memory barrier in `rcu_exit_nohz()` and `rcu_enter_nohz()` modeled in Promela? ■

Answer:
Promela assumes sequential consistency, so it is not necessary to model memory barriers. In fact, one must instead explicitly model lack of memory barriers, for example, as shown in Listing 12.13 on page 227. ❑

Quick Quiz 12.14:
Isn't it a bit strange to model `rcu_exit_nohz()` followed by `rcu_enter_nohz()`? Wouldn't it be more natural to instead model entry before exit? ■

Answer:
It probably would be more natural, but we will need this particular order for the liveness checks that we will add later. ❑

Quick Quiz 12.15:
Wait a minute! In the Linux kernel, both `dynticks_progress_counter` and `rcu_dyntick_snapshot` are per-CPU variables. So why are they instead being modeled as single global variables? ■

Answer:
Because the grace-period code processes each CPU's `dynticks_progress_counter` and `rcu_dyntick_snapshot` variables separately, we can collapse the state onto a single CPU. If the grace-period code were instead to do something special given specific values on specific CPUs, then we would indeed need to model multiple CPUs. But fortunately, we can safely confine ourselves to two CPUs, the one running the grace-period processing and the one entering and leaving dynticks-idle mode. ❑

Quick Quiz 12.16:
Given there are a pair of back-to-back changes to `grace_period_state` on lines 25 and 26, how can we be sure that line 25's changes won't be lost? ■

Answer:
Recall that Promela and Spin trace out every possible sequence of state changes. Therefore, timing is irrelevant: Promela/Spin will be quite happy to jam the entire rest of the model between those two statements unless some state variable specifically prohibits doing so. ❑

Quick Quiz 12.17:
But what would you do if you needed the statements in a single `EXECUTE_MAINLINE()` group to execute non-atomically? ■

Answer:
The easiest thing to do would be to put each such statement in its own `EXECUTE_MAINLINE()` statement. ❑

Quick Quiz 12.18:
But what if the `dynticks_nohz()` process had "if" or "do" statements with conditions, where the statement bodies of these constructs needed to execute non-atomically? ■

Answer:
One approach, as we will see in a later section, is to use explicit labels and "goto" statements. For example, the construct:

```
if
:: i == 0 -> a = -1;
:: else -> a = -2;
fi;
```

could be modeled as something like:

```
EXECUTE_MAINLINE(stmt1,
                 if
                 :: i == 0 -> goto stmt1_then;
                 :: else -> goto stmt1_else;
                 fi)
stmt1_then: skip;
EXECUTE_MAINLINE(stmt1_then1, a = -1; goto stmt1_end)
stmt1_else: skip;
EXECUTE_MAINLINE(stmt1_then1, a = -2)
stmt1_end: skip;
```

However, it is not clear that the macro is helping much in the case of the "if" statement, so these sorts of situations will be open-coded in the following sections. ❑

Quick Quiz 12.19:
Why are lines 46 and 47 (the "`in_dyntick_irq = 0;`" and the "`i++;`") executed atomically? ■

Answer:
These lines of code pertain to controlling the model, not to the code being modeled, so there is no reason to model them non-atomically. The motivation for modeling them atomically is to reduce the size of the state space. ❑

www.dbooks.org

Quick Quiz 12.20:
What property of interrupts is this `dynticks_irq()` process unable to model? ■

Answer:
One such property is nested interrupts, which are handled in the following section. ❑

Quick Quiz 12.21:
Does Paul *always* write his code in this painfully incremental manner? ■

Answer:
Not always, but more and more frequently. In this case, Paul started with the smallest slice of code that included an interrupt handler, because he was not sure how best to model interrupts in Promela. Once he got that working, he added other features. (But if he was doing it again, he would start with a "toy" handler. For example, he might have the handler increment a variable twice and have the mainline code verify that the value was always even.)

Why the incremental approach? Consider the following, attributed to Brian W. Kernighan:

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

This means that any attempt to optimize the production of code should place at least 66 % of its emphasis on optimizing the debugging process, even at the expense of increasing the time and effort spent coding. Incremental coding and testing is one way to optimize the debugging process, at the expense of some increase in coding effort. Paul uses this approach because he rarely has the luxury of devoting full days (let alone weeks) to coding and debugging. ❑

Quick Quiz 12.22:
But what happens if an NMI handler starts running before an IRQ handler completes, and if that NMI handler continues running until a second IRQ handler starts? ■

Answer:
This cannot happen within the confines of a single CPU. The first IRQ handler cannot complete until the NMI handler returns. Therefore, if each of the `dynticks` and `dynticks_nmi` variables have taken on an even value during a given time interval, the corresponding CPU really was in a quiescent state at some time during that interval. ❑

Quick Quiz 12.23:
This is still pretty complicated. Why not just have a `cpumask_t` with per-CPU bits, clearing the bit when entering an IRQ or NMI handler, and setting it upon exit? ■

Answer:
Although this approach would be functionally correct, it would result in excessive IRQ entry/exit overhead on large machines. In contrast, the approach laid out in this section allows each CPU to touch only per-CPU data on IRQ and NMI entry/exit, resulting in much lower IRQ entry/exit overhead, especially on large machines. ❑

Quick Quiz 12.24:
But x86 has strong memory ordering, so why formalize its memory model? ■

Answer:
Actually, academics consider the x86 memory model to be weak because it can allow prior stores to be reordered with subsequent loads. From an academic viewpoint, a strong memory model is one that allows absolutely no reordering, so that all threads agree on the order of all operations visible to them.

Plus it really is the case that developers are sometimes confused about x86 memory ordering. ❑

Quick Quiz 12.25:
Why does line 8 of Listing 12.23 initialize the registers? Why not instead initialize them on lines 4 and 5? ■

Answer:
Either way works. However, in general, it is better to use initialization than explicit instructions. The explicit instructions are used in this example to demonstrate their use. In addition, many of the litmus tests available on the tool's web site (`https://www.cl.cam.ac.uk/~pes20/ppcmem/`) were automatically generated, which generates explicit initialization instructions. ❑

Quick Quiz 12.26:
But whatever happened to line 17 of Listing 12.23, the one that is the `Fail1:` label? ■

Answer:
The implementation of powerpc version of `atomic_add_return()` loops when the `stwcx` instruction fails, which

it communicates by setting non-zero status in the condition-code register, which in turn is tested by the `bne` instruction. Because actually modeling the loop would result in state-space explosion, we instead branch to the Fail: label, terminating the model with the initial value of 2 in P0's `r3` register, which will not trigger the exists assertion.

There is some debate about whether this trick is universally applicable, but I have not seen an example where it fails. ❑

Quick Quiz 12.27:
Does the Arm Linux kernel have a similar bug? ■

Answer:
Arm does not have this particular bug because it places `smp_mb()` before and after the `atomic_add_return()` function's assembly-language implementation. PowerPC no longer has this bug; it has long since been fixed [Her11]. ❑

Quick Quiz 12.28:
Does the `lwsync` on line 10 in Listing 12.23 provide sufficient ordering? ■

Answer:
It depends on the semantics required. The rest of this answer assumes that the assembly language for `P0` in Listing 12.23 is supposed to implement a value-returning atomic operation.

As is discussed in Chapter 15, Linux kernel's memory consistency model requires value-returning atomic RMW operations to be fully ordered on both sides. The ordering provided by `lwsync` is insufficient for this purpose, and so `sync` should be used instead. This change has since been made [Fen15] in response to an email thread discussing a couple of other litmus tests [McK15f]. Finding any other bugs that the Linux kernel might have is left as an exercise for the reader.

In other enviroments providing weaker semantics, `lwsync` might be sufficient. But not for the Linux kernel's value-returning atomic operations! ❑

Quick Quiz 12.29:
What do you have to do to run `herd` on litmus tests like that shown in Listing 12.29? ■

Answer:
Get version v4.17 (or later) of the Linux-kernel source code, then follow the instructions in `tools/memory-model/README` to install the needed tools. Then follow the further instructions to run these tools on the litmus test of your choice. ❑

Table E.4: Locking: Modeling vs. Emulation Time (s)

# Proc.	Model	Emulate			
		`filter`		`exists`	
		`cmpxchg`	`xchg`	`cmpxchg`	`xchg`
2	0.004	0.022	0.027	0.039	0.058
3	0.041	0.743	0.968	1.653	3.203
4	0.374	59.565	74.818	151.962	500.960
5	4.905				

Quick Quiz 12.30:
Why bother modeling locking directly? Why not simply emulate locking with atomic operations? ■

Answer:
In a word, performance, as can be seen in Table E.4. The first column shows the number of herd processes modeled. The second column shows the herd runtime when modeling `spin_lock()` and `spin_unlock()` directly in herd's cat language. The third column shows the herd runtime when emulating `spin_lock()` with `cmpxchg_acquire()` and `spin_unlock()` with `smp_store_release()`, using the herd `filter` clause to reject executions that fail to acquire the lock. The fourth column is like the third, but using `xchg_acquire()` instead of `cmpxchg_acquire()`. The fifth and sixth columns are like the third and fourth, but instead using the herd `exists` clause to reject executions that fail to acquire the lock.

Note also that use of the `filter` clause is about twice as fast as is use of the `exists` clause. This is no surprise because the `filter` clause allows early abandoning of excluded executions, where the executions that are excluded are the ones in which the lock is concurrently held by more than one process.

More important, modeling `spin_lock()` and `spin_unlock()` directly ranges from five times faster to more than two orders of magnitude faster than modeling emulated locking. This should also be no surprise, as direct modeling raises the level of abstraction, thus reducing the number of events that herd must model. Because almost everything that herd does is of exponential computational complexity, modest reductions in the number of events produces exponentially large reductions in runtime.

www.dbooks.org

Thus, in formal verification even more than in parallel programming itself, divide and conquer!!! ❑

Quick Quiz 12.31:
Wait!!! Isn't leaking pointers out of an RCU read-side critical section a critical bug??? ■

Answer:
Yes, it usually is a critical bug. However, in this case, the updater has been cleverly constructed to properly handle such pointer leaks. But please don't make a habit of doing this sort of thing, and especially don't do this without having put a lot of thought into making some more conventional approach work. ❑

Quick Quiz 12.32:
In Listing 12.32, why couldn't a reader fetch `c` just before `P1()` zeroed it on line 45, and then later store this same value back into `c` just after it was zeroed, thus defeating the zeroing operation? ■

Answer:
Because the reader advances to the next element on line 24, thus avoiding storing a pointer to the same element as was fetched. ❑

Quick Quiz 12.33:
In Listing 12.32, why not have just one call to `synchronize_rcu()` immediately before line 48? ■

Answer:
Because this results in `P0()` accessing a freed element. But don't take my word for this, try it out in `herd`! ❑

Quick Quiz 12.34:
Also in Listing 12.32, can't line 48 be `WRITE_ONCE()` instead of `smp_store_release()`? ■

Answer:
That is an excellent question. As of late 2018, the answer is "no one knows". Much depends on the semantics of Armv8's conditional-move instruction. While awaiting clarity on these semantics, `smp_store_release()` is the safe choice. ❑

Quick Quiz 12.35:
But shouldn't sufficiently low-level software be for all intents and purposes immune to being exploited by black hats? ■

Answer:
Unfortunately, no.

At one time, Paul E. McKenny felt that Linux-kernel RCU was immune to such exploits, but the advent of Row Hammer showed him otherwise. After all, if the black hats can hit the system's DRAM, they can hit any and all low-level software, even including RCU.

And in 2018, this possibility passed from the realm of theoretical speculation into the hard and fast realm of objective reality [McK19a]. ❑

Quick Quiz 12.36:
In light of the full verification of the L4 microkernel, isn't this limited view of formal verification just a little bit obsolete? ■

Answer:
Unfortunately, no.

The first full verification of the L4 microkernel was a tour de force, with a large number of Ph.D. students hand-verifying code at a very slow per-student rate. This level of effort could not be applied to most software projects because the rate of change is just too great. Furthermore, although the L4 microkernel is a large software artifact from the viewpoint of formal verification, it is tiny compared to a great number of projects, including LLVM, GCC, the Linux kernel, Hadoop, MongoDB, and a great many others. In addition, this verification did have limits, as the researchers freely admit, to their credit: `https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html#does-sel4-have-zero-bugs`.

Although formal verification is finally starting to show some promise, including more-recent L4 verifications involving greater levels of automation, it currently has no chance of completely displacing testing in the foreseeable future. And although I would dearly love to be proven wrong on this point, please note that such proof will be in the form of a real tool that verifies real software, not in the form of a large body of rousing rhetoric.

Perhaps someday formal verification will be used heavily for validation, including for what is now known as regression testing. Section 17.4 looks at what would be required to make this possibility a reality. ❑

E.13 Putting It All Together

Quick Quiz 13.1:
Why not implement reference-acquisition using a simple compare-and-swap operation that only acquires a reference if the reference counter is non-zero? ■

Answer:
Although this can resolve the race between the release of the last reference and acquisition of a new reference, it does absolutely nothing to prevent the data structure from being freed and reallocated, possibly as some completely different type of structure. It is quite likely that the "simple compare-and-swap operation" would give undefined results if applied to the differently typed structure.

In short, use of atomic operations such as compare-and-swap absolutely requires either type-safety or existence guarantees.

But what if it is absolutely necessary to let the type change?

One approach is for each such type to have the reference counter at the same location, so that as long as the reallocation results in an object from this group of types, all is well. If you do this in C, make sure you comment the reference counter in each structure in which it appears. In C++, use inheritance and templates. ❑

Quick Quiz 13.2:
Why isn't it necessary to guard against cases where one CPU acquires a reference just after another CPU releases the last reference? ■

Answer:
Because a CPU must already hold a reference in order to legally acquire another reference. Therefore, if one CPU releases the last reference, there had better not be any CPU acquiring a new reference! ❑

Quick Quiz 13.3:
Suppose that just after the `atomic_sub_and_test()` on line 22 of Listing 13.2 is invoked, that some other CPU invokes `kref_get()`. Doesn't this result in that other CPU now having an illegal reference to a released object? ■

Answer:
This cannot happen if these functions are used correctly. It is illegal to invoke `kref_get()` unless you already hold a reference, in which case the `kref_sub()` could not possibly have decremented the counter to zero. ❑

Quick Quiz 13.4:
Suppose that `kref_sub()` returns zero, indicating that the `release()` function was not invoked. Under what conditions can the caller rely on the continued existence of the enclosing object? ■

Answer:
The caller cannot rely on the continued existence of the object unless it knows that at least one reference will continue to exist. Normally, the caller will have no way of knowing this, and must therefore carefully avoid referencing the object after the call to `kref_sub()`.

Interested readers are encouraged to work around this limitation using RCU, in particular, `call_rcu()`. ❑

Quick Quiz 13.5:
Why not just pass `kfree()` as the release function? ■

Answer:
Because the `kref` structure normally is embedded in a larger structure, and it is necessary to free the entire structure, not just the `kref` field. This is normally accomplished by defining a wrapper function that does a `container_of()` and then a `kfree()`. ❑

Quick Quiz 13.6:
Why can't the check for a zero reference count be made in a simple "if" statement with an atomic increment in its "then" clause? ■

Answer:
Suppose that the "if" condition completed, finding the reference counter value equal to one. Suppose that a release operation executes, decrementing the reference counter to zero and therefore starting cleanup operations. But now the "then" clause can increment the counter back to a value of one, allowing the object to be used after it has been cleaned up.

This use-after-cleanup bug is every bit as bad as a full-fledged use-after-free bug. ❑

Quick Quiz 13.7:
Why on earth did we need that global lock in the first place? ■

Answer:
A given thread's `__thread` variables vanish when that thread exits. It is therefore necessary to synchronize any operation that accesses other threads' `__thread` variables with thread exit. Without such synchronization, accesses

www.dbooks.org

to __thread variable of a just-exited thread will result in segmentation faults. ❑

Quick Quiz 13.8:
Hey!!! Line 48 of Listing 13.5 modifies a value in a pre-existing countarray structure! Didn't you say that this structure, once made available to read_count(), remained constant??? ■

Answer:
Indeed I did say that. And it would be possible to make count_register_thread() allocate a new structure, much as count_unregister_thread() currently does.

But this is unnecessary. Recall the derivation of the error bounds of read_count() that was based on the snapshots of memory. Because new threads start with initial counter values of zero, the derivation holds even if we add a new thread partway through read_count()'s execution. So, interestingly enough, when adding a new thread, this implementation gets the effect of allocating a new structure, but without actually having to do the allocation.

On the other hand, count_unregister_thread() can result in the outgoing thread's work being double counted. This can happen when read_count() is invoked between lines Line 65 and Line 66. There are efficient ways of avoiding this double-counting, but these are left as an exercise for the reader. ❑

Quick Quiz 13.9:
Given the fixed-size counterp array, exactly how does this code avoid a fixed upper bound on the number of threads??? ■

Answer:
You are quite right, that array does in fact reimpose the fixed upper limit. This limit may be avoided by tracking threads with a linked list, as is done in userspace RCU [DMS+12]. Doing something similar for this code is left as an exercise for the reader. ❑

Quick Quiz 13.10:
Wow! Listing 13.5 contains 70 lines of code, compared to only 42 in Listing 5.4. Is this extra complexity really worth it? ■

Answer:
This of course needs to be decided on a case-by-case basis. If you need an implementation of read_count() that scales linearly, then the lock-based implementation shown in Listing 5.4 simply will not work for you. On the other hand, if calls to read_count() are sufficiently rare, then the lock-based version is simpler and might thus be better, although much of the size difference is due to the structure definition, memory allocation, and NULL return checking.

Listing E.9: Localized Correlated Measurement Fields

```
1 struct measurement {
2         double meas_1;
3         double meas_2;
4         double meas_3;
5 };
6
7 struct animal {
8         char name[40];
9         double age;
10        struct measurement *mp;
11        struct measurement meas;
12        char photo[0]; /* large bitmap. */
13 };
```

Of course, a better question is "Why doesn't the language implement cross-thread access to __thread variables?" After all, such an implementation would make both the locking and the use of RCU unnecessary. This would in turn enable an implementation that was even simpler than the one shown in Listing 5.4, but with all the scalability and performance benefits of the implementation shown in Listing 13.5! ❑

Quick Quiz 13.11:
But cant't the approach shown in Listing 13.9 result in extra cache misses, in turn resulting in additional read-side overhead? ■

Answer:
Indeed it can.

One way to avoid this cache-miss overhead is shown in Listing E.9: Simply embed an instance of a measurement structure named meas into the animal structure, and point the ->mp field at this ->meas field.

Measurement updates can then be carried out as follows:

1. Allocate a new measurement structure and place the new measurements into it.
2. Use rcu_assign_pointer() to point ->mp to this new structure.
3. Wait for a grace period to elapse, for example using either synchronize_rcu() or call_rcu().
4. Copy the measurements from the new measurement structure into the embedded ->meas field.
5. Use rcu_assign_pointer() to point ->mp back to the old embedded ->meas field.

6. After another grace period elapses, free up the new `measurement` structure.

This approach uses a heavier weight update procedure to eliminate the extra cache miss in the common case. The extra cache miss will be incurred only while an update is actually in progress. ❑

Quick Quiz 13.12:
But how does this scan work while a resizable hash table is being resized? In that case, neither the old nor the new hash table is guaranteed to contain all the elements in the hash table! ■

Answer:
True, resizable hash tables as described in Section 10.4 cannot be fully scanned while being resized. One simple way around this is to acquire the `hashtab` structure's `->ht_lock` while scanning, but this prevents more than one scan from proceeding concurrently.

Another approach is for updates to mutate the old hash table as well as the new one while resizing is in progress. This would allow scans to find all elements in the old hash table. Implementing this is left as an exercise for the reader. ❑

E.14 Advanced Synchronization

Quick Quiz 14.1:
So why not ditch antique languages like C and C++ for something more modern? ■

Answer:
That won't help unless the more-modern languages proponents are energetic enough to write their own compiler backends. The usual practice of re-using existing backends also reuses charming properties such as refusal to support pointers to lifetime-ended objects. ❑

Quick Quiz 14.2:
But what about battery-powered systems? They don't require energy flowing into the system as a whole. ■

Answer:
Sooner or later, the battery must be recharged, which requires energy to flow into the system. ❑

Quick Quiz 14.3:
But given the results from queueing theory, won't low utilization merely improve the average response time rather than improving the worst-case response time? And isn't worst-case response time all that most real-time systems really care about? ■

Answer:
Yes, but . . .

Those queueing-theory results assume infinite "calling populations", which in the Linux kernel might correspond to an infinite number of tasks. As of early 2021, no real system supports an infinite number of tasks, so results assuming infinite calling populations should be expected to have less-than-infinite applicability.

Other queueing-theory results have *finite* calling populations, which feature sharply bounded response times [HL86]. These results better model real systems, and these models do predict reductions in both average and worst-case response times as utilizations decrease. These results can be extended to model concurrent systems that use synchronization mechanisms such as locking [Bra11, SM04a].

In short, queueing-theory results that accurately describe real-world real-time systems show that worst-case response time decreases with decreasing utilization. ❑

Quick Quiz 14.4:
Formal verification is already quite capable, benefiting from decades of intensive study. Are additional advances *really* required, or is this just a practitioner's excuse to continue to lazily ignore the awesome power of formal verification? ■

Answer:
Perhaps this situation is just a theoretician's excuse to avoid diving into the messy world of real software? Perhaps more constructively, the following advances are required:

1. Formal verification needs to handle larger software artifacts. The largest verification efforts have been for systems of only about 10,000 lines of code, and those have been verifying much simpler properties than real-time latencies.

2. Hardware vendors will need to publish formal timing guarantees. This used to be common practice back when hardware was much simpler, but today's complex hardware results in excessively complex expressions for worst-case performance. Unfortunately,

www.dbooks.org

energy-efficiency concerns are pushing vendors in the direction of even more complexity.

3. Timing analysis needs to be integrated into development methodologies and IDEs.

All that said, there is hope, given recent work formalizing the memory models of real computer systems [AMP+11, AKNT13]. On the other hand, formal verification has just as much trouble as does testing with the astronomical number of variants of the Linux kernel that can be constructed from different combinations of its tens of thousands of Kconfig options. Sometimes life is hard! ❑

Quick Quiz 14.5:
Differentiating real-time from non-real-time based on what can "be achieved straightforwardly by non-real-time systems and applications" is a travesty! There is absolutely no theoretical basis for such a distinction!!! Can't we do better than that??? ■

Answer:
This distinction is admittedly unsatisfying from a strictly theoretical perspective. But on the other hand, it is exactly what the developer needs in order to decide whether the application can be cheaply and easily developed using standard non-real-time approaches, or whether the more difficult and expensive real-time approaches are required. In other words, although theory is quite important, for those of us called upon to complete practical projects, theory supports practice, never the other way around. ❑

Quick Quiz 14.6:
But if you only allow one reader at a time to read-acquire a reader-writer lock, isn't that the same as an exclusive lock??? ■

Answer:
Indeed it is, other than the API. And the API is important because it allows the Linux kernel to offer real-time capabilities without having the -rt patchset grow to ridiculous sizes.

However, this approach clearly and severely limits read-side scalability. The Linux kernel's -rt patchset was long able to live with this limitation for several reasons: (1) Real-time systems have traditionally been relatively small, (2) Real-time systems have generally focused on process control, thus being unaffected by scalability limitations in the I/O subsystems, and (3) Many of the Linux kernel's reader-writer locks have been converted to RCU.

However, the day came when it was absolutely necessary to permit concurrent readers, as described in the text following this quiz. ❑

Quick Quiz 14.7:
Suppose that preemption occurs just after the load from `t->rcu_read_unlock_special.s` on line 12 of Listing 14.3. Mightn't that result in the task failing to invoke `rcu_read_unlock_special()`, thus failing to remove itself from the list of tasks blocking the current grace period, in turn causing that grace period to extend indefinitely? ■

Answer:
That is a real problem, and it is solved in RCU's scheduler hook. If that scheduler hook sees that the value of `t->rcu_read_lock_nesting` is negative, it invokes `rcu_read_unlock_special()` if needed before allowing the context switch to complete. ❑

Quick Quiz 14.8:
But isn't correct operation despite fail-stop bugs a valuable fault-tolerance property? ■

Answer:
Yes and no.

Yes in that non-blocking algorithms can provide fault tolerance in the face of fail-stop bugs, but no in that this is grossly insufficient for practical fault tolerance. For example, suppose you had a wait-free queue, and further suppose that a thread has just dequeued an element. If that thread now succumbs to a fail-stop bug, the element it has just dequeued is effectively lost. True fault tolerance requires way more than mere non-blocking properties, and is beyond the scope of this book. ❑

Quick Quiz 14.9:
I couldn't help but spot the word "includes" before this list. Are there other constraints? ■

Answer:
Indeed there are, and lots of them. However, they tend to be specific to a given situation, and many of them can be thought of as refinements of some of the constraints listed above. For example, the many constraints on choices of data structure will help meeting the "Bounded time spent in any given critical section" constraint. ❑

Quick Quiz 14.10:
Given that real-time systems are often used for safety-

critical applications, and given that runtime memory allocation is forbidden in many safety-critical situations, what is with the call to `malloc()`??? ■

Answer:
In early 2016, situations forbidding runtime memory were also not at all interested in multithreaded computing. So the runtime memory allocation is not an additional obstacle to safety criticality.

However, by 2020 runtime memory allocation in multicore real-time systems was gaining some traction. ❑

Quick Quiz 14.11:
Don't you need some kind of synchronization to protect `update_cal()`? ■

Answer:
Indeed you do, and you could use any of a number of techniques discussed earlier in this book. One of those techniques is use of a single updater thread, which would result in exactly the code shown in `update_cal()` in Listing 14.6. ❑

E.15 Advanced Synchronization: Memory Ordering

Quick Quiz 15.1:
This chapter has been rewritten since the first edition. Did memory ordering change all *that* since 2014? ■

Answer:
The earlier memory-ordering section had its roots in a pair of Linux Journal articles [McK05a, McK05b] dating back to 2005. Since then, the C and C++ memory models [Bec11] have been formalized (and critiqued [BS14, BD14, VBC+15, BMN+15, LVK+17, BGV17]), executable formal memory models for computer systems have become the norm [MSS12, McK11d, SSA+11, AMP+11, AKNT13, AKT13, AMT14, MS14, FSP+17, ARM17], and there is even a memory model for the Linux kernel [AMM+17a, AMM+17b, AMM+18], along with a paper describing differences between the C11 and Linux memory models [MWPF18].

Given all this progress since 2005, it was high time for a full rewrite! ❑

Quick Quiz 15.2:
The compiler can also reorder Thread `P0()`'s and Thread `P1()`'s memory accesses in Listing 15.1, right? ■

Answer:
In general, compiler optimizations carry out more extensive and profound reorderings than CPUs can. However, in this case, the volatile accesses in `READ_ONCE()` and `WRITE_ONCE()` prevent the compiler from reordering. And also from doing much else as well, so the examples in this section will be making heavy use of `READ_ONCE()` and `WRITE_ONCE()`. See Section 15.3 for more detail on the need for `READ_ONCE()` and `WRITE_ONCE()`. ❑

Quick Quiz 15.3:
But wait!!! On row 2 of Table 15.1 both `x0` and `x1` each have two values at the same time, namely zero and two. How can that possibly work??? ■

Answer:
There is an underlying cache-coherence protocol that straightens things out, which are discussed in Appendix C.2. But if you think that a given variable having two values at the same time is surprising, just wait until you get to Section 15.2.1! ❑

Quick Quiz 15.4:
But don't the values also need to be flushed from the cache to main memory? ■

Answer:
Perhaps surprisingly, not necessarily! On some systems, if the two variables are being used heavily, they might be bounced back and forth between the CPUs' caches and never land in main memory. ❑

Quick Quiz 15.5:
The rows in Table 15.3 seem quite random and confused. Whatever is the conceptual basis of this table??? ■

Answer:
The rows correspond roughly to hardware mechanisms of increasing power and overhead.

The `WRITE_ONCE()` row captures the fact that accesses to a single variable are always fully ordered, as indicated by the "SV"column. Note that all other operations providing ordering against accesses to multiple variables also provide this same-variable ordering.

www.dbooks.org

The READ_ONCE() row captures the fact that (as of 2021) compilers and CPUs do not indulge in user-visible speculative stores, so that any store whose address, data, or execution depends on a prior load is guaranteed to happen after that load completes. However, this guarantee assumes that these dependencies have been constructed carefully, as described in Sections 15.3.2 and 15.3.3.

The "_relaxed() RMW operation" row captures the fact that a value-returning _relaxed() RMW has done a load and a store, which are every bit as good as a READ_ONCE() and a WRITE_ONCE(), respectively.

The *_dereference() row captures the address and data dependency ordering provided by rcu_dereference() and friends. Again, these dependencies must been constructed carefully, as described in Sections 15.3.2 and 15.3.3.

The "Successful *_acquire()" row captures the fact that many CPUs have special "acquire" forms of loads and of atomic RMW instructions, and that many other CPUs have lightweight memory-barrier instructions that order prior loads against subsequent loads and stores.

The "Successful *_release()" row captures the fact that many CPUs have special "release" forms of stores and of atomic RMW instructions, and that many other CPUs have lightweight memory-barrier instructions that order prior loads and stores against subsequent stores.

The smp_rmb() row captures the fact that many CPUs have lightweight memory-barrier instructions that order prior loads against subsequent loads. Similarly, the smp_wmb() row captures the fact that many CPUs have lightweight memory-barrier instructions that order prior stores against subsequent stores.

None of the ordering operations thus far require prior stores to be ordered against subsequent loads, which means that these operations need not interfere with store buffers, whose main purpose in life is in fact to reorder prior stores against subsequent loads. The lightweight nature of these operations is precisely due to their policy of store-buffer non-interference. However, as noted earlier, it is sometimes necessary to interfere with the store buffer in order to prevent prior stores from being reordered against later stores, which brings us to the remaining rows in this table.

The smp_mb() row corresponds to the full memory barrier available on most platforms, with Itanium being the exception that proves the rule. However, even on Itanium, smp_mb() provides full ordering with respect to READ_ONCE() and WRITE_ONCE(), as discussed in Section 15.5.4.

The "Successful full-strength non-void RMW" row captures the fact that on some platforms (such as x86) atomic RMW instructions provide full ordering both before and after. The Linux kernel therefore requires that full-strength non-void atomic RMW operations provide full ordering in cases where these operations succeed. (Full-strength atomic RMW operation's names do not end in _relaxed, _acquire, or _release.) As noted earlier, the case where these operations do not succeed is covered by the "_relaxed() RMW operation" row.

However, the Linux kernel does not require that either void or _relaxed() atomic RMW operations provide any ordering whatsoever, with the canonical example being atomic_inc(). Therefore, these operations, along with failing non-void atomic RMW operations may be preceded by smp_mb__before_atomic() and followed by smp_mb__after_atomic() to provide full ordering for any accesses preceding or following both. No ordering need be provided for accesses between the smp_mb__before_atomic() (or, similarly, the smp_mb__after_atomic()) and the atomic RMW operation, as indicated by the "a" entries on the smp_mb__before_atomic() and smp_mb__after_atomic() rows of the table.

In short, the structure of this table is dictated by the properties of the underlying hardware, which are constrained by nothing other than the laws of physics, which were covered back in Chapter 3. That is, the table is not random, although it is quite possible that you are confused. ❑

Quick Quiz 15.6:
Why is Table 15.3 missing smp_mb__after_unlock_lock() and smp_mb__after_spinlock()? ■

Answer:
These two primitives are rather specialized, and at present seem difficult to fit into Table 15.3. The smp_mb__after_unlock_lock() primitive is intended to be placed immediately after a lock acquisition, and ensures that all CPUs see all accesses in prior critical sections as happening before all accesses following the smp_mb__after_unlock_lock() and also before all accesses in later critical sections. Here "all CPUs" includes those CPUs not holding that lock, and "prior critical sections" includes all prior critical sections for the lock in question as well as all prior critical sections for all other locks that were released by the same CPU that executed the smp_mb__after_unlock_lock().

The smp_mb__after_spinlock() provides the same guarantees as does smp_mb__after_unlock_lock(),

but also provides additional visibility guarantees for other accesses performed by the CPU that executed the `smp_mb__after_spinlock()`. Given any store S performed prior to any earlier lock acquisition and any load L performed after the `smp_mb__after_spinlock()`, all CPUs will see S as happening before L. In other words, if a CPU performs a store S, acquires a lock, executes an `smp_mb__after_spinlock()`, then performs a load L, all CPUs will see S as happening before L. ❑

Quick Quiz 15.7:
But how can I know that a given project can be designed and coded within the confines of these rules of thumb? ■

Answer:
Much of the purpose of the remainder of this chapter is to answer exactly that question! ❑

Quick Quiz 15.8:
How can you tell which memory barriers are strong enough for a given use case? ■

Answer:
Ah, that is a deep question whose answer requires most of the rest of this chapter. But the short answer is that `smp_mb()` is almost always strong enough, albeit at some cost. ❑

Quick Quiz 15.9:
Wait!!! Where do I find this tooling that automatically analyzes litmus tests??? ■

Answer:
Get version v4.17 (or later) of the Linux-kernel source code, then follow the instructions in `tools/memory-model/README` to install the needed tools. Then follow the further instructions to run these tools on the litmus test of your choice. ❑

Quick Quiz 15.10:
What assumption is the code fragment in Listing 15.3 making that might not be valid on real hardware? ■

Answer:
The code assumes that as soon as a given CPU stops seeing its own value, it will immediately see the final agreed-upon value. On real hardware, some of the CPUs might well see several intermediate results before converging on the final value. The actual code used to produce the data in the figures discussed later in this section was therefore somewhat more complex. ❑

Quick Quiz 15.11:
How could CPUs possibly have different views of the value of a single variable *at the same time?* ■

Answer:
As discussed in Section 15.1.1, many CPUs have store buffers that record the values of recent stores, which do not become globally visible until the corresponding cache line makes its way to the CPU. Therefore, it is quite possible for each CPU to see its own value for a given variable (in its own store buffer) at a single point in time—and for main memory to hold yet another value. One of the reasons that memory barriers were invented was to allow software to deal gracefully with situations like this one.

Fortunately, software rarely cares about the fact that multiple CPUs might see multiple values for the same variable. ❑

Quick Quiz 15.12:
Why do CPUs 2 and 3 come to agreement so quickly, when it takes so long for CPUs 1 and 4 to come to the party? ■

Answer:
CPUs 2 and 3 are a pair of hardware threads on the same core, sharing the same cache hierarchy, and therefore have very low communications latencies. This is a NUMA, or, more accurately, a NUCA effect.

This leads to the question of why CPUs 2 and 3 ever disagree at all. One possible reason is that they each might have a small amount of private cache in addition to a larger shared cache. Another possible reason is instruction reordering, given the short 10-nanosecond duration of the disagreement and the total lack of memory-ordering operations in the code fragment. ❑

Quick Quiz 15.13:
But why make load-load reordering visible to the user? Why not just use speculative execution to allow execution to proceed in the common case where there are no intervening stores, in which case the reordering cannot be visible anyway? ■

Answer:
They can and many do, otherwise systems containing strongly ordered CPUs would be slow indeed. However, speculative execution does have its downsides, especially

www.dbooks.org

if speculation must be rolled back frequently, particularly on battery-powered systems. But perhaps future systems will be able to overcome these disadvantages. Until then, we can expect vendors to continue producing weakly ordered CPUs. ❑

Quick Quiz 15.14:
Why should strongly ordered systems pay the performance price of unnecessary `smp_rmb()` and `smp_wmb()` invocations? Shouldn't weakly ordered systems shoulder the full cost of their misordering choices??? ■

Answer:
That is in fact exactly what happens. On strongly ordered systems, `smp_rmb()` and `smp_wmb()` emit no instructions, but instead just constrain the compiler. Thus, in this case, weakly ordered systems do in fact shoulder the full cost of their memory-ordering choices. ❑

Quick Quiz 15.15:
But how do we know that *all* platforms really avoid triggering the `exists` clauses in Listings 15.10 and 15.11? ■

Answer:
Answering this requires identifying three major groups of platforms: (1) Total-store-order (TSO) platforms, (2) Weakly ordered platforms, and (3) DEC Alpha.

The TSO platforms order all pairs of memory references except for prior stores against later loads. Because the address dependency on lines 18 and 19 of Listing 15.10 is instead a load followed by another load, TSO platforms preserve this address dependency. They also preserve the address dependency on lines 17 and 18 of Listing 15.11 because this is a load followed by a store. Because address dependencies must start with a load, TSO platforms implicitly but completely respect them, give or take compiler optimizations, hence the need for `READ_ONCE()`.

Weakly ordered platforms don't necessarily maintain ordering of unrelated accesses. However, the address dependencies in Listings 15.10 and 15.11 are not unrelated: There is an address dependency. The hardware tracks dependencies and maintains the needed ordering.

There is one (famous) exception to this rule for weakly ordered platforms, and that exception is DEC Alpha for load-to-load address dependencies. And this is why, in Linux kernels predating v4.15, DEC Alpha requires the explicit memory barrier supplied for it by the now-obsolete `lockless_dereference()` on line 18 of Listing 15.10. However, DEC Alpha does track load-to-store address dependencies, which is why line 17 of Listing 15.11 does not need a `lockless_dereference()`, even in Linux kernels predating v4.15.

To sum up, current platforms either respect address dependencies implicitly, as is the case for TSO platforms (x86, mainframe, SPARC, . . .), have hardware tracking for address dependencies (Arm, PowerPC, MIPS, . . .), have the required memory barriers supplied by `READ_ONCE()` (DEC Alpha in Linux kernel v4.15 and later), or supplied by `rcu_dereference()` (DEC Alpha in Linux kernel v4.14 and earlier). ❑

Quick Quiz 15.16:
SP, MP, LB, and now S. Where do all these litmus-test abbreviations come from and how can anyone keep track of them? ■

Answer:
The best scorecard is the infamous `test6.pdf` [SSA+11]. Unfortunately, not all of the abbreviations have catchy expansions like SB (store buffering), MP (message passing), and LB (load buffering), but at least the list of abbreviations is readily available. ❑

Quick Quiz 15.17:
But wait!!! Line 17 of Listing 15.12 uses `READ_ONCE()`, which marks the load as volatile, which means that the compiler absolutely must emit the load instruction even if the value is later multiplied by zero. So how can the compiler possibly break this data dependency? ■

Answer:
Yes, the compiler absolutely must emit a load instruction for a volatile load. But if you multiply the value loaded by zero, the compiler is well within its rights to substitute a constant zero for the result of that multiplication, which will break the data dependency on many platforms.

Worse yet, if the dependent store does not use `WRITE_ONCE()`, the compiler could hoist it above the load, which would cause even TSO platforms to fail to provide ordering. ❑

Quick Quiz 15.18:
Wouldn't control dependencies be more robust if they were mandated by language standards??? ■

Answer:
But of course! And perhaps in the fullness of time they will be so mandated. ❑

Listing E.10: Litmus Test Distinguishing Multicopy Atomic From Other Multicopy Atomic

```
1  C C-MP-OMCA+o-o-o+o-rmb-o
2
3  {}
4
5  P0(int *x, int *y)
6  {
7    int r0;
8
9    WRITE_ONCE(*x, 1);
10   r0 = READ_ONCE(*x);
11   WRITE_ONCE(*y, r0);
12 }
13
14 P1(int *x, int *y)
15 {
16   int r1;
17   int r2;
18
19   r1 = READ_ONCE(*y);
20   smp_rmb();
21   r2 = READ_ONCE(*x);
22 }
23
24 exists (1:r1=1 /\ 1:r2=0)
```

Quick Quiz 15.19:
But in Listing 15.15, wouldn't be just as bad if `P2()`'s `r1` and `r2` obtained the values 2 and 1, respectively, while `P3()`'s `r3` and `r4` obtained the values 1 and 2, respectively? ■

Answer:
Yes, it would. Feel free to modify the `exists` clause to check for that outcome and see what happens. ❑

Quick Quiz 15.20:
Can you give a specific example showing different behavior for multicopy atomic on the one hand and other-multicopy atomic on the other? ■

Answer:
Listing E.10 (`C-MP-OMCA+o-o-o+o-rmb-o.litmus`) shows such a test.

On a multicopy-atomic platform, `P0()`'s store to `x` on line 9 must become visible to both `P0()` and `P1()` simultaneously. Because this store becomes visible to `P0()` on line 10, before `P0()`'s store to `y` on line 11, `P0()`'s store to `x` must become visible before its store to `y` everywhere, including `P1()`. Therefore, if `P1()`'s load from `y` on line 19 returns the value 1, so must its load from `x` on line 21, given that the `smp_rmb()` on line 20 forces these two loads to execute in order. Therefore, the `exists` clause on line 24 cannot trigger on a multicopy-atomic platform.

In contrast, on an other-multicopy-atomic platform, `P0()` could see its own store early, so that there would be no constraint on the order of visibility of the two stores from to `P1()`, which in turn allows the `exists` clause to trigger. ❑

Quick Quiz 15.21:
Then who would even *think* of designing a system with shared store buffers??? ■

Answer:
This is in fact a very natural design for any system having multiple hardware threads per core. Natural from a hardware point of view, that is! ❑

Quick Quiz 15.22:
But just how is it fair that `P0()` and `P1()` must share a store buffer and a cache, but `P2()` gets one each of its very own??? ■

Answer:
Presumably there is a `P3()`, as is in fact shown in Figure 15.8, that shares `P2()`'s store buffer and cache. But not necessarily. Some platforms allow different cores to disable different numbers of threads, allowing the hardware to adjust to the needs of the workload at hand. For example, a single-threaded critical-path portion of the workload might be assigned to a core with only one thread enabled, thus allowing the single thread running that portion of the workload to use the entire capabilities of that core. Other more highly parallel but cache-miss-prone portions of the workload might be assigned to cores with all hardware threads enabled to provide improved throughput. This improved throughput could be due to the fact that while one hardware thread is stalled on a cache miss, the other hardware threads can make forward progress.

In such cases, performance requirements override quaint human notions of fairness. ❑

Quick Quiz 15.23:
Referring to Table 15.4, why on earth would `P0()`'s store take so long to complete when `P1()`'s store complete so quickly? In other words, does the `exists` clause on line 28 of Listing 15.16 really trigger on real systems? ■

Answer:
You need to face the fact that it really can trigger. Akira Yokosawa used the `litmus7` tool to run this litmus test on a POWER8 system. Out of 1,000,000,000 runs, 4 triggered

www.dbooks.org

Listing E.11: R Litmus Test With Write Memory Barrier (No Ordering)

```
1 C C-R+o-wmb-o+o-mb-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7   WRITE_ONCE(*x0, 1);
8   smp_wmb();
9   WRITE_ONCE(*x1, 1);
10 }
11
12 P1(int *x0, int *x1)
13 {
14   int r2;
15
16   WRITE_ONCE(*x1, 2);
17   smp_mb();
18   r2 = READ_ONCE(*x0);
19 }
20
21 exists (1:r2=0 /\ x1=2)
```

the `exists` clause. Thus, triggering the `exists` clause is not merely a one-in-a-million occurrence, but rather a one-in-a-hundred-million occurrence. But it nevertheless really does trigger on real systems. ❑

Quick Quiz 15.24:
But it is not necessary to worry about propagation unless there are at least three threads in the litmus test, right? ■

Answer:
Wrong.

Listing E.11 (`C-R+o-wmb-o+o-mb-o.litmus`) shows a two-thread litmus test that requires propagation due to the fact that it only has store-to-store and load-to-store links between its pair of threads. Even though `P0()` is fully ordered by the `smp_wmb()` and `P1()` is fully ordered by the `smp_mb()`, the counter-temporal nature of the links means that the `exists` clause on line 21 really can trigger. To prevent this triggering, the `smp_wmb()` on line 8 must become an `smp_mb()`, bringing propagation into play twice, once for each non-temporal link. ❑

Quick Quiz 15.25:
But given that `smp_mb()` has the propagation property, why doesn't the `smp_mb()` on line 25 of Listing 15.18 prevent the `exists` clause from triggering? ■

Answer:
As a rough rule of thumb, the `smp_mb()` barrier's propagation property is sufficient to maintain ordering through only one load-to-store link between processes. Unfortunately, Listing 15.18 has not one but two load-to-store links, with the first being from the `READ_ONCE()` on line 17 to the `WRITE_ONCE()` on line 24 and the second being from the `READ_ONCE()` on line 26 to the `WRITE_ONCE()` on line 7. Therefore, preventing the `exists` clause from triggering should be expected to require not one but two instances of `smp_mb()`.

As a special exception to this rule of thumb, a release-acquire chain can have one load-to-store link between processes and still prohibit the cycle. ❑

Listing E.12: 2+2W Litmus Test (No Ordering)

```
1 C C-2+2W+o-o+o-o
2
3 {}
4
5 P0(int *x0, int *x1)
6 {
7   WRITE_ONCE(*x0, 1);
8   WRITE_ONCE(*x1, 2);
9 }
10
11 P1(int *x0, int *x1)
12 {
13   WRITE_ONCE(*x1, 1);
14   WRITE_ONCE(*x0, 2);
15 }
16
17 exists (x0=1 /\ x1=1)
```

Quick Quiz 15.26:
But for litmus tests having only ordered stores, as shown in Listing 15.20 (`C-2+2W+o-wmb-o+o-wmb-o.litmus`), research shows that the cycle is prohibited, even in weakly ordered systems such as Arm and Power [SSA+11]. Given that, are store-to-store really *always* counter-temporal??? ■

Answer:
This litmus test is indeed a very interesting curiosity. Its ordering apparently occurs naturally given typical weakly ordered hardware design, which would normally be considered a great gift from the relevant laws of physics and cache-coherency-protocol mathematics.

Unfortunately, no one has been able to come up with a software use case for this gift that does not have a much better alternative implementation. Therefore, neither the C11 nor the Linux kernel memory models provide any guarantee corresponding to Listing 15.20. This means that the `exists` clause on line 19 can trigger.

Of course, without the barrier, there are no ordering guarantees, even on real weakly ordered hardware, as shown in Listing E.12 (`C-2+2W+o-o+o-o.litmus`). ❑

Listing E.13: LB Litmus Test With No Acquires

```
1  C C-LB+o-data-o+o-data-o+o-data-o
2
3  {
4  x1=1;
5  x2=2;
6  }
7
8  P0(int *x0, int *x1)
9  {
10   int r2;
11
12   r2 = READ_ONCE(*x0);
13   WRITE_ONCE(*x1, r2);
14 }
15
16 P1(int *x1, int *x2)
17 {
18   int r2;
19
20   r2 = READ_ONCE(*x1);
21   WRITE_ONCE(*x2, r2);
22 }
23
24 P2(int *x2, int *x0)
25 {
26   int r2;
27
28   r2 = READ_ONCE(*x2);
29   WRITE_ONCE(*x0, r2);
30 }
31
32 exists (0:r2=2 /\ 1:r2=0 /\ 2:r2=1)
```

Quick Quiz 15.27:
Can you construct a litmus test like that in Listing 15.21 that uses *only* dependencies? ■

Answer:
Listing E.13 shows a somewhat nonsensical but very real example. Creating a more useful (but still real) litmus test is left as an exercise for the reader. ❑

Quick Quiz 15.28:
Suppose we have a short release-acquire chain along with one load-to-store link and one store-to-store link, like that shown in Listing 15.25. Given that there is only one of each type of non-store-to-load link, the `exists` cannot trigger, right? ■

Answer:
Wrong. It is the number of non-store-to-load links that matters. If there is only one non-store-to-load link, a release-acquire chain can prevent the `exists` clause from triggering. However, if there is more than one non-store-to-load link, be they store-to-store, load-to-store, or any combination thereof, it is necessary to have at least one full barrier (`smp_mb()` or better) between each non-store-to-load link. In Listing 15.25, preventing the `exists` clause from triggering therefore requires an additional full barrier between either `P0()`'s or `P1()`'s accesses. ❑

Quick Quiz 15.29:
There are store-to-load links, load-to-store links, and store-to-store links. But what about load-to-load links? ■

Answer:
The problem with the concept of load-to-load links is that if the two loads from the same variable return the same value, there is no way to determine their ordering. The only way to determine their ordering is if they return different values, in which case there had to have been an intervening store. And that intervening store means that there is no load-to-load link, but rather a load-to-store link followed by a store-to-load link. ❑

Quick Quiz 15.30:
Why not place a `barrier()` call immediately before a plain store to prevent the compiler from inventing stores? ■

Answer:
Because it would not work. Although the compiler would be prevented from inventing a store prior to the `barrier()`, nothing would prevent it from inventing a store between that `barrier()` and the plain store. ❑

Quick Quiz 15.31:
Why can't you simply dereference the pointer before comparing it to `&reserve_int` on line 6 of Listing 15.26? ■

Answer:
For first, it might be necessary to invoke `handle_reserve()` before `do_something_with()`.

But more relevant to memory ordering, the compiler is often within its rights to hoist the comparison ahead of the dereferences, which would allow the compiler to use `&reserve_int` instead of the variable `p` that the hardware has tagged with a dependency. ❑

Quick Quiz 15.32:
But it should be safe to compare two pointer variables, right? After all, the compiler doesn't know the value of either, so how can it possibly learn anything from the comparison? ■

Answer:
Unfortunately, the compiler really can learn enough to

www.dbooks.org

Listing E.14: Breakable Dependencies With Non-Constant Comparisons

```
1 int *gp1;
2 int *p;
3 int *q;
4
5 p = rcu_dereference(gp1);
6 q = get_a_pointer();
7 if (p == q)
8   handle_equality(p);
9 do_something_with(*p);
```

Listing E.15: Broken Dependencies With Non-Constant Comparisons

```
1 int *gp1;
2 int *p;
3 int *q;
4
5 p = rcu_dereference(gp1);
6 q = get_a_pointer();
7 if (p == q) {
8   handle_equality(q);
9   do_something_with(*q);
10 } else {
11   do_something_with(*p);
12 }
```

break your dependency chain, for example, as shown in Listing E.14. The compiler is within its rights to transform this code into that shown in Listing E.15, and might well make this transformation due to register pressure if `handle_equality()` was inlined and needed a lot of registers. Line 9 of this transformed code uses `q`, which although equal to `p`, is not necessarily tagged by the hardware as carrying a dependency. Therefore, this transformed code does not necessarily guarantee that line 9 is ordered after line 5.[12] ❑

Quick Quiz 15.33:
But doesn't the condition in line 35 supply a control dependency that would keep line 36 ordered after line 34? ■

Answer:
Yes, but no. Yes, there is a control dependency, but control dependencies do not order later loads, only later stores. If you really need ordering, you could place an `smp_rmb()` between lines 35 and 36. Or better yet, have `update()` allocate two structures instead of reusing the structure. For more information, see Section 15.3.3. ❑

Quick Quiz 15.34:
Can't you instead add an `smp_mb()` to `P1()` in Listing 15.30? ■

[12] Kudos to Linus Torvalds for providing this example.

Answer:
Not given the Linux kernel memory model. (Try it!) However, you can instead replace `P0()`'s `WRITE_ONCE()` with `smp_store_release()`, which usually has less overhead than does adding an `smp_mb()`. ❑

Quick Quiz 15.35:
But doesn't PowerPC have weak unlock-lock ordering properties within the Linux kernel, allowing a write before the unlock to be reordered with a read after the lock? ■

Answer:
Yes, but only from the perspective of a third thread not holding that lock. In contrast, memory allocators need only concern themselves with the two threads migrating the memory. It is after all the developer's responsibility to properly synchronize with any other threads that need access to the newly migrated block of memory. ❑

Quick Quiz 15.36:
Wait a minute! In QSBR implementations of RCU, no code is emitted for `rcu_read_lock()` and `rcu_read_unlock()`. This means that the RCU read-side critical section in Listing 15.38 isn't just empty, it is completely nonexistent!!! So how can something that doesn't exist at all possibly have any effect whatsoever on ordering??? ■

Answer:
Because in QSBR, RCU read-side critical sections don't actually disappear. Instead, they are extended in both directions until a quiescent state is encountered. For example, in the Linux kernel, the critical section might be expanded to a pair of `schedule()` calls, with one preceding and the other following the critical section. Of course, in non-QSBR implementations, `rcu_read_lock()` and `rcu_read_unlock()` really do emit code, which can clearly provide ordering. And within the Linux kernel, even the QSBR implementation has a compiler `barrier()` in `rcu_read_lock()` and `rcu_read_unlock()`, which is necessary to prevent the compiler from moving memory accesses that might result in page faults into the RCU read-side critical section.

Therefore, strange though it might seem, empty RCU read-side critical sections really can and do provide some degree of ordering. ❑

Quick Quiz 15.37:
Can `P1()`'s accesses be reordered in the litmus tests

shown in Listings 15.36, 15.37, and 15.38 in the same way that they were reordered going from Listing 15.31 to Listing 15.32? ■

Answer:
No, because none of these later litmus tests have more than one access within their RCU read-side critical sections. But what about swapping the accesses, for example, in Listing 15.36, placing `P1()`'s `WRITE_ONCE()` within its critical section and the `READ_ONCE()` before its critical section?

Swapping the accesses allows both instances of `r2` to have a final value of zero, in other words, although RCU read-side critical sections' ordering properties can extend outside of those critical sections, the same is not true of their reordering properties. Checking this with `herd` and explaining why is left as an exercise for the reader. ❑

Quick Quiz 15.38:
What would happen if the `smp_mb()` was instead added between `P2()`'s accesses in Listing 15.40? ■

Answer:
The cycle would again be forbidden. Further analysis is left as an exercise for the reader. ❑

Quick Quiz 15.39:
What happens to code between an atomic operation and an `smp_mb__after_atomic()`? ■

Answer:
First, please don't do this!

But if you do, this intervening code will either be ordered after the atomic operation or before the `smp_mb__after_atomic()`, depending on the architecture, but not both. This also applies to `smp_mb__before_atomic()` and `smp_mb__after_spinlock()`, that is, both the uncertain ordering of the intervening code and the plea to avoid such code. ❑

Quick Quiz 15.40:
Why does Alpha's `READ_ONCE()` include an `mb()` rather than `rmb()`? ■

Answer:
Alpha has only `mb` and `wmb` instructions, so `smp_rmb()` would be implemented by the Alpha `mb` instruction in either case. In addition, at the time that the Linux kernel started relying on dependency ordering, it was not clear that Alpha ordered dependent stores, and thus `smp_mb()` was therefore the safe choice.

However, given the aforementioned v5.9 changes to `READ_ONCE()` and a few of Alpha's atomic read-modify-write operations, no Linux-kernel core code need concern itself with DEC Alpha, thus greatly reducing Paul E. McKenney's incentive to remove Alpha support from the kernel. ❑

Quick Quiz 15.41:
Isn't DEC Alpha significant as having the weakest possible memory ordering? ■

Answer:
Although DEC Alpha does take considerable flak, it does avoid reordering reads from the same CPU to the same variable. It also avoids the out-of-thin-air problem that plagues the Java and C11 memory models [BD14, BMN+15, BS14, Boe20, Gol19, Jef14, MB20, MJST16, Š11, VBC+15]. ❑

Quick Quiz 15.42:
Given that hardware can have a half memory barrier, why don't locking primitives allow the compiler to move memory-reference instructions into lock-based critical sections? ■

Answer:
In fact, as we saw in Section 15.5.3 and will see in Section 15.5.6, hardware really does implement partial memory-ordering instructions and it also turns out that these really are used to construct locking primitives. However, these locking primitives use full compiler barriers, thus preventing the compiler from reordering memory-reference instructions both out of and into the corresponding critical section.

To see why the compiler is forbidden from doing reordering that is permitted by hardware, consider the following sample code in Listing E.16. This code is based on the userspace RCU update-side code [DMS+12, Supplementary Materials Figure 5].

Suppose that the compiler reordered lines 27 and 28 into the critical section starting at line 29. Now suppose that two updaters start executing `synchronize_rcu()` at about the same time. Then consider the following sequence of events:

1. CPU 0 acquires the lock at line 29.

www.dbooks.org

Listing E.16: Userspace RCU Code Reordering

```
1  static inline int rcu_gp_ongoing(unsigned long *ctr)
2  {
3    unsigned long v;
4
5    v = LOAD_SHARED(*ctr);
6    return v && (v != rcu_gp_ctr);
7  }
8
9  static void update_counter_and_wait(void)
10 {
11   struct rcu_reader *index;
12
13   STORE_SHARED(rcu_gp_ctr, rcu_gp_ctr + RCU_GP_CTR);
14   barrier();
15   list_for_each_entry(index, &registry, node) {
16     while (rcu_gp_ongoing(&index->ctr))
17       msleep(10);
18   }
19 }
20
21 void synchronize_rcu(void)
22 {
23   unsigned long was_online;
24
25   was_online = rcu_reader.ctr;
26   smp_mb();
27   if (was_online)
28     STORE_SHARED(rcu_reader.ctr, 0);
29   mutex_lock(&rcu_gp_lock);
30   update_counter_and_wait();
31   mutex_unlock(&rcu_gp_lock);
32   if (was_online)
33     STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
34   smp_mb();
35 }
```

2. Line 27 determines that CPU 0 was online, so it clears its own counter at line 28. (Recall that lines 27 and 28 have been reordered by the compiler to follow line 29).

3. CPU 0 invokes `update_counter_and_wait()` from 30.

4. CPU 0 invokes `rcu_gp_ongoing()` on itself at line 16, and line 5 sees that CPU 0 is in a quiescent state. Control therefore returns to `update_counter_and_wait()`, and line 15 advances to CPU 1.

5. CPU 1 invokes `synchronize_rcu()`, but because CPU 0 already holds the lock, CPU 1 blocks waiting for this lock to become available. Because the compiler reordered lines 27 and 28 to follow 29, CPU 1 does not clear its own counter, despite having been online.

6. CPU 0 invokes `rcu_gp_ongoing()` on CPU 1 at line 16, and line 5 sees that CPU 1 is not in a quiescent state. The `while` loop at line 16 therefore never exits.

So the compiler's reordering results in a deadlock. In contrast, hardware reordering is temporary, so that CPU 1 might undertake its first attempt to acquire the mutex on line 29 before executing lines 27 and 28, but it will eventually execute lines 27 and 28. Because hardware reordering only results in a short delay, it can be tolerated. On the other hand, because compiler reordering results in a deadlock, it must be prohibited.

Some research efforts have used hardware transactional memory to allow compilers to safely reorder more aggressively, but the overhead of hardware transactions has thus far made such optimizations unattractive. ❑

Quick Quiz 15.43:
Why is it necessary to use heavier-weight ordering for load-to-store and store-to-store links, but not for store-to-load links? What on earth makes store-to-load links so special??? ■

Answer:
Recall that load-to-store and store-to-store links can be counter-temporal, as illustrated by Figures 15.10 and 15.11 in Section 15.2.7.2. This counter-temporal nature of load-to-store and store-to-store links necessitates strong ordering.

In constrast, store-to-load links are temporal, as illustrated by Listings 15.12 and 15.13. This temporal nature of store-to-load links permits use of minimal ordering. ❑

E.16 Ease of Use

Quick Quiz 16.1:
Can a similar algorithm be used when deleting elements? ■

Answer:
Yes. However, since each thread must hold the locks of three consecutive elements to delete the middle one, if there are N threads, there must be $2N + 1$ elements (rather than just $N + 1$) in order to avoid deadlock. ❑

Quick Quiz 16.2:
Yetch! What ever possessed someone to come up with an algorithm that deserves to be shaved as much as this one does??? ■

Answer:
That would be Paul.

He was considering the *Dining Philosopher's Problem*, which involves a rather unsanitary spaghetti dinner attended by five philosophers. Given that there are five plates and but five forks on the table, and given that each philosopher requires two forks at a time to eat, one is supposed to come up with a fork-allocation algorithm that avoids deadlock. Paul's response was "Sheesh! Just get five more forks!"

This in itself was OK, but Paul then applied this same solution to circular linked lists.

This would not have been so bad either, but he had to go and tell someone about it! ❑

Quick Quiz 16.3:
Give an exception to this rule. ■

Answer:
One exception would be a difficult and complex algorithm that was the only one known to work in a given situation. Another exception would be a difficult and complex algorithm that was nonetheless the simplest of the set known to work in a given situation. However, even in these cases, it may be very worthwhile to spend a little time trying to come up with a simpler algorithm! After all, if you managed to invent the first algorithm to do some task, it shouldn't be that hard to go on to invent a simpler one. ❑

E.17 Conflicting Visions of the Future

Quick Quiz 17.1:
But suppose that an application exits while holding a `pthread_mutex_lock()` that happens to be located in a file-mapped region of memory? ■

Answer:
Indeed, in this case the lock would persist, much to the consternation of other processes attempting to acquire this lock that is held by a process that no longer exists. Which is why great care is required when using `pthread_mutex` objects located in file-mapped memory regions. ❑

Quick Quiz 17.2:
What about non-persistent primitives represented by data structures in `mmap()` regions of memory? What happens when there is an `exec()` within a critical section of such a primitive? ■

Answer:
If the `exec()`ed program maps those same regions of memory, then this program could in principle simply release the lock. The question as to whether this approach is sound from a software-engineering viewpoint is left as an exercise for the reader. ❑

Quick Quiz 17.3:
MV-RLU looks pretty good! Doesn't it beat RCU hands down? ■

Answer:
One might get that impression from a quick read of the abstract, but more careful readers will notice the "for a wide range of workloads" phrase in the last sentence. It turns out that this phrase is quite important:

1. Their RCU evaluation uses synchronous grace periods, which needlessly throttle updates, as noted in their Section 6.2.1. See Figure 10.11 on Page 182 of this book to see that the venerable asynchronous `call_rcu()` primitive enables RCU to perform and scale quite well with large numbers of updaters. Furthermore, in Section 3.7 of their paper, the authors admit that asynchronous grace periods are important to MV-RLU scalability. A fair comparison would also allow RCU the benefits of asynchrony.

2. They use a poorly tuned 1,000-bucket hash table containing 10,000 elements. In addition, their 448 hardware threads need considerably more than 1,000 buckets to avoid the lock contention that they correctly state limits RCU performance in their benchmarks. A useful comparison would feature a properly tuned hash table.

3. Their RCU hash table used per-bucket locks, which they call out as a bottleneck, which is not a surprise given the long hash chains and small ratio of buckets to threads. A number of their competing mechanisms instead use lockfree techniques, thus avoiding the per-bucket-lock bottleneck, which cynics might claim sheds some light on the authors' otherwise inexplicable choice of poorly tuned hash tables. The first graph in the middle row of the authors' Figure 4 show what RCU can achieve if not hobbled by artificial bottlenecks, as does the first portion of the second graph in that same row.

4. Their linked-list operation permits RLU to do concurrent modifications of different elements in the

www.dbooks.org

list, while RCU is forced to serialize updates. Again, RCU has always worked just fine in conjunction with lockless updaters, a fact that has been set forth in academic literature that the authors cited [DMS+12]. A fair comparison would use the same style of update for RCU as it does for MV-RLU.

5. The authors fail to consider combining RCU and sequence locking, which is used in the Linux kernel to give readers coherent views of multi-pointer updates.

6. The authors fail to consider RCU-based solutions to the Issaquah Challenge [McK16], which also gives readers a coherent view of multi-pointer updates, albeit with a weaker view of "coherent".

It is surprising that the anonymous reviewers of this paper did not demand an apples-to-apples comparison of MV-RCU and RCU. Nevertheless, the authors should be congratulated on producing an academic paper that presents an all-too-rare example of good scalability combined with strong read-side coherence. They are also to be congratulated on overcoming the traditional academic prejudice against asynchronous grace periods, which greatly aided their scalability.

Interestingly enough, RLU and RCU take different approaches to avoid the inherent limitations of STM noted by Hagit Attiya et al. [AHM09]. RCU avoids providing strict serializability and RLU avoids providing invisible read-only transactions, both thus avoiding the limitations. ❑

Quick Quiz 17.4:
Given things like `spin_trylock()`, how does it make any sense at all to claim that TM introduces the concept of failure??? ■

Answer:
When using locking, `spin_trylock()` is a choice, with a corresponding failure-free choice being `spin_lock()`, which is used in the common case, as in there are more than 100 times as many calls to `spin_lock()` than to `spin_trylock()` in the v5.11 Linux kernel. When using TM, the only failure-free choice is the irrevocable transaction, which is not used in the common case. In fact, the irrevocable transaction is not even available in all TM implementations. ❑

Quick Quiz 17.5:
What is to learn? Why not just use TM for memory-based data structures and locking for those rare cases featuring the many silly corner cases listed in this silly section??? ■

Answer:
The year 2005 just called, and it says that it wants its incandescent TM marketing hype back.

In the year 2021, TM still has significant proving to do, even with the advent of HTM, which is covered in the upcoming Section 17.3. ❑

Quick Quiz 17.6:
Why would it matter that oft-written variables shared the cache line with the lock variable? ■

Answer:
If the lock is in the same cacheline as some of the variables that it is protecting, then writes to those variables by one CPU will invalidate that cache line for all the other CPUs. These invalidations will generate large numbers of conflicts and retries, perhaps even degrading performance and scalability compared to locking. ❑

Quick Quiz 17.7:
Why are relatively small updates important to HTM performance and scalability? ■

Answer:
The larger the updates, the greater the probability of conflict, and thus the greater probability of retries, which degrade performance. ❑

Quick Quiz 17.8:
How could a red-black tree possibly efficiently enumerate all elements of the tree regardless of choice of synchronization mechanism??? ■

Answer:
In many cases, the enumeration need not be exact. In these cases, hazard pointers or RCU may be used to protect readers, which provides low probability of conflict with any given insertion or deletion. ❑

Quick Quiz 17.9:
But why can't a debugger emulate single stepping by setting breakpoints at successive lines of the transaction, relying on the retry to retrace the steps of the earlier instances of the transaction? ■

Answer:
This scheme might work with reasonably high probability,

but it can fail in ways that would be quite surprising to most users. To see this, consider the following transaction:

```
1 begin_trans();
2 if (a) {
3   do_one_thing();
4   do_another_thing();
5 } else {
6   do_a_third_thing();
7   do_a_fourth_thing();
8 }
9 end_trans();
```

Suppose that the user sets a breakpoint at line 4, which triggers, aborting the transaction and entering the debugger. Suppose that between the time that the breakpoint triggers and the debugger gets around to stopping all the threads, some other thread sets the value of a to zero. When the poor user attempts to single-step the program, surprise! The program is now in the else-clause instead of the then-clause.

This is *not* what I call an easy-to-use debugger. ❑

Quick Quiz 17.10:
But why would *anyone* need an empty lock-based critical section??? ■

Answer:
See the answer to Quick Quiz 7.18 in Section 7.2.1.

However, it is claimed that given a strongly atomic HTM implementation without forward-progress guarantees, any memory-based locking design based on empty critical sections will operate correctly in the presence of transactional lock elision. Although I have not seen a proof of this statement, there is a straightforward rationale for this claim. The main idea is that in a strongly atomic HTM implementation, the results of a given transaction are not visible until after the transaction completes successfully. Therefore, if you can see that a transaction has started, it is guaranteed to have already completed, which means that a subsequent empty lock-based critical section will successfully "wait" on it—after all, there is no waiting required.

This line of reasoning does not apply to weakly atomic systems (including many STM implementation), and it also does not apply to lock-based programs that use means other than memory to communicate. One such means is the passage of time (for example, in hard real-time systems) or flow of priority (for example, in soft real-time systems).

Locking designs that rely on priority boosting are of particular interest. ❑

Quick Quiz 17.11:
Can't transactional lock elision trivially handle locking's time-based messaging semantics by simply choosing not to elide empty lock-based critical sections? ■

Answer:
It could do so, but this would be both unnecessary and insufficient.

It would be unnecessary in cases where the empty critical section was due to conditional compilation. Here, it might well be that the only purpose of the lock was to protect data, so eliding it completely would be the right thing to do. In fact, leaving the empty lock-based critical section would degrade performance and scalability.

On the other hand, it is possible for a non-empty lock-based critical section to be relying on both the data-protection and time-based and messaging semantics of locking. Using transactional lock elision in such a case would be incorrect, and would result in bugs. ❑

Quick Quiz 17.12:
Given modern hardware [MOZ09], how can anyone possibly expect parallel software relying on timing to work? ■

Answer:
The short answer is that on commonplace commodity hardware, synchronization designs based on any sort of fine-grained timing are foolhardy and cannot be expected to operate correctly under all conditions.

That said, there are systems designed for hard real-time use that are much more deterministic. In the (very unlikely) event that you are using such a system, here is a toy example showing how time-based synchronization can work. Again, do *not* try this on commodity microprocessors, as they have highly nondeterministic performance characteristics.

This example uses multiple worker threads along with a control thread. Each worker thread corresponds to an outbound data feed, and records the current time (for example, from the `clock_gettime()` system call) in a per-thread `my_timestamp` variable after executing each unit of work. The real-time nature of this example results in the following set of constraints:

1. It is a fatal error for a given worker thread to fail to update its timestamp for a time period of more than `MAX_LOOP_TIME`.

2. Locks are used sparingly to access and update global state.

www.dbooks.org

3. Locks are granted in strict FIFO order within a given thread priority.

When worker threads complete their feed, they must disentangle themselves from the rest of the application and place a status value in a per-thread `my_status` variable that is initialized to −1. Threads do not exit; they instead are placed on a thread pool to accommodate later processing requirements. The control thread assigns (and re-assigns) worker threads as needed, and also maintains a histogram of thread statuses. The control thread runs at a real-time priority no higher than that of the worker threads.

Worker threads' code is as follows:

```
1  int my_status = -1;  /* Thread local. */
2
3  while (continue_working()) {
4    enqueue_any_new_work();
5    wp = dequeue_work();
6    do_work(wp);
7    my_timestamp = clock_gettime(...);
8  }
9
10 acquire_lock(&departing_thread_lock);
11
12 /*
13  * Disentangle from application, might
14  * acquire other locks, can take much longer
15  * than MAX_LOOP_TIME, especially if many
16  * threads exit concurrently.
17  */
18 my_status = get_return_status();
19 release_lock(&departing_thread_lock);
20
21 /* thread awaits repurposing. */
```

The control thread's code is as follows:

```
1  for (;;) {
2    for_each_thread(t) {
3      ct = clock_gettime(...);
4      d = ct - per_thread(my_timestamp, t);
5      if (d >= MAX_LOOP_TIME) {
6        /* thread departing. */
7        acquire_lock(&departing_thread_lock);
8        release_lock(&departing_thread_lock);
9        i = per_thread(my_status, t);
10       status_hist[i]++; /* Bug if TLE! */
11     }
12   }
13   /* Repurpose threads as needed. */
14 }
```

Line 5 uses the passage of time to deduce that the thread has exited, executing lines 6 and 10 if so. The empty lock-based critical section on lines 7 and 8 guarantees that any thread in the process of exiting completes (remember that locks are granted in FIFO order!).

Once again, do not try this sort of thing on commodity microprocessors. After all, it is difficult enough to get this right on systems specifically designed for hard real-time use! ❑

Quick Quiz 17.13:
But the `boostee()` function in Listing 17.1 alternatively acquires its locks in reverse order! Won't this result in deadlock? ■

Answer:
No deadlock will result. To arrive at deadlock, two different threads must each acquire the two locks in opposite orders, which does not happen in this example. However, deadlock detectors such as lockdep [Cor06a] will flag this as a false positive. ❑

Quick Quiz 17.14:
So a bunch of people set out to supplant locking, and they mostly end up just optimizing locking??? ■

Answer:
At least they accomplished something useful! And perhaps there will continue to be additional HTM progress over time [SNGK17, SBN+20, GGK18, PMDY20]. ❑

Quick Quiz 17.15:
Given the groundbreaking nature of the various verifiers used in the SEL4 project, why doesn't this chapter cover them in more depth? ■

Answer:
There can be no doubt that the verifiers used by the SEL4 project are quite capable. However, SEL4 started as a single-CPU project. And although SEL4 has gained multi-processor capabilities, it is currently using very coarse-grained locking that is similar to the Linux kernel's old Big Kernel Lock (BKL). There will hopefully come a day when it makes sense to add SEL4's verifiers to a book on parallel programming, but this is not yet that day. ❑

Quick Quiz 17.16:
Why bother with a separate `filter` command on line 27 of Listing 17.2 instead of just adding the condition to the `exists` clause? And wouldn't it be simpler to use `xchg_acquire()` instead of `cmpxchg_acquire()`? ■

Answer:
The `filter` clause causes the `herd` tool to discard executions at an earlier stage of processing than does the `exists` clause, which provides significant speedups.

As for `xchg_acquire()`, this atomic operation will do a write whether or not lock acquisition succeeds, which means that a model using `xchg_acquire()` will have more operations than one using `cmpxchg_acquire()`,

Table E.5: Emulating Locking: Performance Comparison (s)

		cmpxchg_acquire()		xchg_acquire()	
#	Lock	filter	exists	filter	exists
2	0.004	0.022	0.039	0.027	0.058
3	0.041	0.743	1.653	0.968	3.203
4	0.374	59.565	151.962	74.818	500.96
5	4.905				

which won't do a write in the failed-acquisition case. More writes means more combinatorial to explode, as shown in Table E.5 (`C-SB+l-o-o-u+l-o-o-*u.litmus`, `C-SB+l-o-o-u+l-o-o-u*-C.litmus`, `C-SB+l-o-o-u+l-o-o-u*-CE.litmus`, `C-SB+l-o-o-u+l-o-o-u*-X.litmus`, and `C-SB+l-o-o-u+l-o-o-u*-XE.litmus`). This table clearly shows that `cmpxchg_acquire()` outperforms `xchg_acquire()` and that use of the `filter` clause outperforms use of the `exists` clause. ❑

Quick Quiz 17.17:
How do we know that the MTBFs of known bugs is a good estimate of the MTBFs of bugs that have not yet been located? ■

Answer:
We don't, but it does not matter.

To see this, note that the 7 % figure only applies to injected bugs that were subsequently located: It necessarily ignores any injected bugs that were never found. Therefore, the MTBF statistics of known bugs is likely to be a good approximation of that of the injected bugs that are subsequently located.

A key point in this whole section is that we should be more concerned about bugs that inconvenience users than about other bugs that never actually manifest. This of course is *not* to say that we should completely ignore bugs that have not yet inconvenienced users, just that we should properly prioritize our efforts so as to fix the most important and urgent bugs first. ❑

Quick Quiz 17.18:
But the formal-verification tools should immediately find all the bugs introduced by the fixes, so why is this a problem? ■

Answer:
It is a problem because real-world formal-verification tools (as opposed to those that exist only in the imaginations of the more vociferous proponents of formal verification) are not omniscient, and thus are only able to locate certain types of bugs. For but one example, formal-verification tools are unlikely to spot a bug corresponding to an omitted assertion or, equivalently, a bug corresponding to an undiscovered portion of the specification. ❑

Quick Quiz 17.19:
But many formal-verification tools can only find one bug at a time, so that each bug must be fixed before the tool can locate the next. How can bug-fix efforts be prioritized given such a tool? ■

Answer:
One approach is to provide a simple fix that might not be suitable for a production environment, but which allows the tool to locate the next bug. Another approach is to restrict configuration or inputs so that the bugs located thus far cannot occur. There are a number of similar approaches, but the common theme is that fixing the bug from the tool's viewpoint is usually much easier than constructing and validating a production-quality fix, and the key point is to prioritize the larger efforts required to construct and validate the production-quality fixes. ❑

Quick Quiz 17.20:
How would testing stack up in the scorecard shown in Table 17.5? ■

Answer:
It would be blue all the way down, with the possible exception of the third row (overhead) which might well be marked down for testing's difficulty finding improbable bugs.

On the other hand, improbable bugs are often also irrelevant bugs, so your mileage may vary.

Much depends on the size of your installed base. If your code is only ever going to run on (say) 10,000 systems, Murphy can actually be a really nice guy. Everything that can go wrong, will. Eventually. Perhaps in geologic time.

But if your code is running on 20 billion systems, like the Linux kernel was said to in late 2017, Murphy can be a real jerk! Everything that can go wrong, will, and it can go wrong really quickly!!! ❑

www.dbooks.org

Quick Quiz 17.21:
But aren't there a great many more formal-verification systems than are shown in Table 17.5? ■

Answer:
Indeed there are! This table focuses on those that Paul has used, but others are proving to be useful. Formal verification has been heavily used in the seL4 project [SM13], and its tools can now handle modest levels of concurrency. More recently, Catalin Marinas used Lamport's TLA tool [Lam02] to locate some forward-progress bugs in the Linux kernel's queued spinlock implementation. Will Deacon fixed these bugs [Dea18], and Catalin verified Will's fixes [Mar18].

Lighter-weight formal verification tools have been used heavily in production [LBD+04, BBC+10, Coo18, SAE+18, DFLO19]. ❑

E.18 Important Questions

Quick Quiz A.1:
What SMP coding errors can you see in these examples? See `time.c` for full code. ■

Answer:

1. Missing barrier() or volatile on tight loops.
2. Missing memory barriers on update side.
3. Lack of synchronization between producer and consumer. ❑

Quick Quiz A.2:
How could there be such a large gap between successive consumer reads? See `timelocked.c` for full code. ■

Answer:

1. The consumer might be preempted for long time periods.
2. A long-running interrupt might delay the consumer.
3. Cache misses might delay the consumer.
4. The producer might also be running on a faster CPU than is the consumer (for example, one of the CPUs might have had to decrease its clock frequency due to heat-dissipation or power-consumption constraints). ❑

Quick Quiz A.3:
Suppose a portion of a program uses RCU read-side primitives as its only synchronization mechanism. Is this parallelism or concurrency? ■

Answer:
Yes. ❑

Quick Quiz A.4:
In what part of the second (scheduler-based) perspective would the lock-based single-thread-per-CPU workload be considered "concurrent"? ■

Answer:
The people who would like to arbitrarily subdivide and interleave the workload. Of course, an arbitrary subdivision might end up separating a lock acquisition from the corresponding lock release, which would prevent any other thread from acquiring that lock. If the locks were pure spinlocks, this could even result in deadlock. ❑

Quick Quiz A.5:
But if fully ordered implementations cannot offer stronger guarantees that the better performing and more scalable weakly ordered imptementations, why bother with full ordering? ■

Answer:
Because strongly ordered implementations are sometimes able to provide greater consistency among sets of calls to functions accessing a given data structure. For example, compare the atomic counter of Listing 5.2 to the statistical counter of Section 5.2. Suppose that one thread is adding the value 3 and another is adding the value 5, while two other threads are concurrently reading the counter's value. With atomic counters, it is not possible for one of the readers to obtain the value 3 while the other obtains the value 5. With statistical counters, this outcome really can happen. In fact, in some computing environments, this outcome can happen even on relatively strongly ordered hardware such as x86.

Therefore, if your user happen to need this admittedly unusual level of consistency, you should avoid weakly ordered statistical counters. ❑

Listing E.17: Deadlock in Lock-Based RCU Implementation

```
1  void foo(void)
2  {
3    spin_lock(&my_lock);
4    rcu_read_lock();
5    do_something();
6    rcu_read_unlock();
7    do_something_else();
8    spin_unlock(&my_lock);
9  }
10
11 void bar(void)
12 {
13   rcu_read_lock();
14   spin_lock(&my_lock);
15   do_some_other_thing();
16   spin_unlock(&my_lock);
17   do_whatever();
18   rcu_read_unlock();
19 }
```

E.19 "Toy" RCU Implementations

Quick Quiz B.1:
Why wouldn't any deadlock in the RCU implementation in Listing B.1 also be a deadlock in any other RCU implementation? ■

Answer:
Suppose the functions `foo()` and `bar()` in Listing E.17 are invoked concurrently from different CPUs. Then `foo()` will acquire `my_lock()` on line 3, while `bar()` will acquire `rcu_gp_lock` on line 13.

When `foo()` advances to line 4, it will attempt to acquire `rcu_gp_lock`, which is held by `bar()`. Then when `bar()` advances to line 14, it will attempt to acquire `my_lock`, which is held by `foo()`.

Each function is then waiting for a lock that the other holds, a classic deadlock.

Other RCU implementations neither spin nor block in `rcu_read_lock()`, hence avoiding deadlocks. ❑

Quick Quiz B.2:
Why not simply use reader-writer locks in the RCU implementation in Listing B.1 in order to allow RCU readers to proceed in parallel? ■

Answer:
One could in fact use reader-writer locks in this manner. However, textbook reader-writer locks suffer from memory contention, so that the RCU read-side critical sections would need to be quite long to actually permit parallel execution [McK03].

On the other hand, use of a reader-writer lock that is read-acquired in `rcu_read_lock()` would avoid the deadlock condition noted above. ❑

Quick Quiz B.3:
Wouldn't it be cleaner to acquire all the locks, and then release them all in the loop from lines 15–18 of Listing B.2? After all, with this change, there would be a point in time when there were no readers, simplifying things greatly. ■

Answer:
Making this change would re-introduce the deadlock, so no, it would not be cleaner. ❑

Quick Quiz B.4:
Is the implementation shown in Listing B.2 free from deadlocks? Why or why not? ■

Answer:
One deadlock is where a lock is held across `synchronize_rcu()`, and that same lock is acquired within an RCU read-side critical section. However, this situation could deadlock any correctly designed RCU implementation. After all, the `synchronize_rcu()` primitive must wait for all pre-existing RCU read-side critical sections to complete, but if one of those critical sections is spinning on a lock held by the thread executing the `synchronize_rcu()`, we have a deadlock inherent in the definition of RCU.

Another deadlock happens when attempting to nest RCU read-side critical sections. This deadlock is peculiar to this implementation, and might be avoided by using recursive locks, or by using reader-writer locks that are read-acquired by `rcu_read_lock()` and write-acquired by `synchronize_rcu()`.

However, if we exclude the above two cases, this implementation of RCU does not introduce any deadlock situations. This is because only time some other thread's lock is acquired is when executing `synchronize_rcu()`, and in that case, the lock is immediately released, prohibiting a deadlock cycle that does not involve a lock held across the `synchronize_rcu()` which is the first case above. ❑

Quick Quiz B.5:
Isn't one advantage of the RCU algorithm shown in Listing B.2 that it uses only primitives that are widely available, for example, in POSIX pthreads? ■

www.dbooks.org

Answer:
This is indeed an advantage, but do not forget that `rcu_dereference()` and `rcu_assign_pointer()` are still required, which means `volatile` manipulation for `rcu_dereference()` and memory barriers for `rcu_assign_pointer()`. Of course, many Alpha CPUs require memory barriers for both primitives. ❑

Quick Quiz B.6:
But what if you hold a lock across a call to `synchronize_rcu()`, and then acquire that same lock within an RCU read-side critical section? ■

Answer:
Indeed, this would deadlock any legal RCU implementation. But is `rcu_read_lock()` *really* participating in the deadlock cycle? If you believe that it is, then please ask yourself this same question when looking at the RCU implementation in Appendix B.9. ❑

Quick Quiz B.7:
How can the grace period possibly elapse in 40 nanoseconds when `synchronize_rcu()` contains a 10-millisecond delay? ■

Answer:
The update-side test was run in absence of readers, so the `poll()` system call was never invoked. In addition, the actual code has this `poll()` system call commented out, the better to evaluate the true overhead of the update-side code. Any production uses of this code would be better served by using the `poll()` system call, but then again, production uses would be even better served by other implementations shown later in this section. ❑

Quick Quiz B.8:
Why not simply make `rcu_read_lock()` wait when a concurrent `synchronize_rcu()` has been waiting too long in the RCU implementation in Listing B.3? Wouldn't that prevent `synchronize_rcu()` from starving? ■

Answer:
Although this would in fact eliminate the starvation, it would also mean that `rcu_read_lock()` would spin or block waiting for the writer, which is in turn waiting on readers. If one of these readers is attempting to acquire a lock that the spinning/blocking `rcu_read_lock()` holds, we again have deadlock.

In short, the cure is worse than the disease. See Appendix B.4 for a proper cure. ❑

Quick Quiz B.9:
Why the memory barrier on line 5 of `synchronize_rcu()` in Listing B.6 given that there is a spin-lock acquisition immediately after? ■

Answer:
The spin-lock acquisition only guarantees that the spin-lock's critical section will not "bleed out" to precede the acquisition. It in no way guarantees that code preceding the spin-lock acquisition won't be reordered into the critical section. Such reordering could cause a removal from an RCU-protected list to be reordered to follow the complementing of `rcu_idx`, which could allow a newly starting RCU read-side critical section to see the recently removed data element.

Exercise for the reader: use a tool such as Promela/spin to determine which (if any) of the memory barriers in Listing B.6 are really needed. See Chapter 12 for information on using these tools. The first correct and complete response will be credited. ❑

Quick Quiz B.10:
Why is the counter flipped twice in Listing B.6? Shouldn't a single flip-and-wait cycle be sufficient? ■

Answer:
Both flips are absolutely required. To see this, consider the following sequence of events:

1. Line 8 of `rcu_read_lock()` in Listing B.5 picks up `rcu_idx`, finding its value to be zero.

2. Line 8 of `synchronize_rcu()` in Listing B.6 complements the value of `rcu_idx`, setting its value to one.

3. Lines 10–12 of `synchronize_rcu()` find that the value of `rcu_refcnt[0]` is zero, and thus returns. (Recall that the question is asking what happens if lines 13–20 are omitted.)

4. Lines 9 and 10 of `rcu_read_lock()` store the value zero to this thread's instance of `rcu_read_idx` and increments `rcu_refcnt[0]`, respectively. Execution then proceeds into the RCU read-side critical section.

5. Another instance of `synchronize_rcu()` again complements `rcu_idx`, this time setting its value to zero. Because `rcu_refcnt[1]` is zero, `synchronize_rcu()` returns immediately. (Recall that `rcu_read_lock()` incremented `rcu_refcnt[0]`, not `rcu_refcnt[1]`!)

6. The grace period that started in step 5 has been allowed to end, despite the fact that the RCU read-side critical section that started beforehand in step 4 has not completed. This violates RCU semantics, and could allow the update to free a data element that the RCU read-side critical section was still referencing.

Exercise for the reader: What happens if `rcu_read_lock()` is preempted for a very long time (hours!) just after line 8? Does this implementation operate correctly in that case? Why or why not? The first correct and complete response will be credited. ❑

Quick Quiz B.11:
Given that atomic increment and decrement are so expensive, why not just use non-atomic increment on line 10 and a non-atomic decrement on line 25 of Listing B.5? ■

Answer:
Using non-atomic operations would cause increments and decrements to be lost, in turn causing the implementation to fail. See Appendix B.5 for a safe way to use non-atomic operations in `rcu_read_lock()` and `rcu_read_unlock()`. ❑

Quick Quiz B.12:
Come off it! We can see the `atomic_read()` primitive in `rcu_read_lock()`!!! So why are you trying to pretend that `rcu_read_lock()` contains no atomic operations??? ■

Answer:
The `atomic_read()` primitives does not actually execute atomic machine instructions, but rather does a normal load from an `atomic_t`. Its sole purpose is to keep the compiler's type-checking happy. If the Linux kernel ran on 8-bit CPUs, it would also need to prevent "store tearing", which could happen due to the need to store a 16-bit pointer with two eight-bit accesses on some 8-bit systems. But thankfully, it seems that no one runs Linux on 8-bit systems. ❑

Quick Quiz B.13:
Great, if we have N threads, we can have $2N$ ten-millisecond waits (one set per `flip_counter_and_wait()` invocation, and even that assumes that we wait only once for each thread. Don't we need the grace period to complete *much* more quickly? ■

Answer:
Keep in mind that we only wait for a given thread if that thread is still in a pre-existing RCU read-side critical section, and that waiting for one hold-out thread gives all the other threads a chance to complete any pre-existing RCU read-side critical sections that they might still be executing. So the only way that we would wait for $2N$ intervals would be if the last thread still remained in a pre-existing RCU read-side critical section despite all the waiting for all the prior threads. In short, this implementation will not wait unnecessarily.

However, if you are stress-testing code that uses RCU, you might want to comment out the `poll()` statement in order to better catch bugs that incorrectly retain a reference to an RCU-protected data element outside of an RCU read-side critical section. ❑

Quick Quiz B.14:
All of these toy RCU implementations have either atomic operations in `rcu_read_lock()` and `rcu_read_unlock()`, or `synchronize_rcu()` overhead that increases linearly with the number of threads. Under what circumstances could an RCU implementation enjoy lightweight implementations for all three of these primitives, all having deterministic ($O(1)$) overheads and latencies? ■

Answer:
Special-purpose uniprocessor implementations of RCU can attain this ideal [McK09a]. ❑

Quick Quiz B.15:
If any even value is sufficient to tell `synchronize_rcu()` to ignore a given task, why don't lines 11 and 12 of Listing B.14 simply assign zero to `rcu_reader_gp`? ■

Answer:
Assigning zero (or any other even-numbered constant) would in fact work, but assigning the value of `rcu_gp_ctr` can provide a valuable debugging aid, as it gives the developer an idea of when the corresponding thread last exited an RCU read-side critical section. ❑

www.dbooks.org

Quick Quiz B.16:
Why are the memory barriers on lines 19 and 31 of Listing B.14 needed? Aren't the memory barriers inherent in the locking primitives on lines 20 and 30 sufficient? ■

Answer:
These memory barriers are required because the locking primitives are only guaranteed to confine the critical section. The locking primitives are under absolutely no obligation to keep other code from bleeding in to the critical section. The pair of memory barriers are therefore requires to prevent this sort of code motion, whether performed by the compiler or by the CPU. ❑

Quick Quiz B.17:
Couldn't the update-side batching optimization described in Appendix B.6 be applied to the implementation shown in Listing B.14? ■

Answer:
Indeed it could, with a few modifications. This work is left as an exercise for the reader. ❑

Quick Quiz B.18:
Is the possibility of readers being preempted in lines 3–4 of Listing B.14 a real problem, in other words, is there a real sequence of events that could lead to failure? If not, why not? If so, what is the sequence of events, and how can the failure be addressed? ■

Answer:
It is a real problem, there is a sequence of events leading to failure, and there are a number of possible ways of addressing it. For more details, see the Quick Quizzes near the end of Appendix B.8. The reason for locating the discussion there is to (1) give you more time to think about it, and (2) because the nesting support added in that section greatly reduces the time required to overflow the counter. ❑

Quick Quiz B.19:
Why not simply maintain a separate per-thread nesting-level variable, as was done in previous section, rather than having all this complicated bit manipulation? ■

Answer:
The apparent simplicity of the separate per-thread variable is a red herring. This approach incurs much greater complexity in the guise of careful ordering of operations, especially if signal handlers are to be permitted to contain RCU read-side critical sections. But don't take my word for it, code it up and see what you end up with! ❑

Quick Quiz B.20:
Given the algorithm shown in Listing B.16, how could you double the time required to overflow the global `rcu_gp_ctr`? ■

Answer:
One way would be to replace the magnitude comparison on lines 32 and 33 with an inequality check of the per-thread `rcu_reader_gp` variable against `rcu_gp_ctr+RCU_GP_CTR_BOTTOM_BIT`. ❑

Quick Quiz B.21:
Again, given the algorithm shown in Listing B.16, is counter overflow fatal? Why or why not? If it is fatal, what can be done to fix it? ■

Answer:
It can indeed be fatal. To see this, consider the following sequence of events:

1. Thread 0 enters `rcu_read_lock()`, determines that it is not nested, and therefore fetches the value of the global `rcu_gp_ctr`. Thread 0 is then preempted for an extremely long time (before storing to its per-thread `rcu_reader_gp` variable).
2. Other threads repeatedly invoke `synchronize_rcu()`, so that the new value of the global `rcu_gp_ctr` is now `RCU_GP_CTR_BOTTOM_BIT` less than it was when thread 0 fetched it.
3. Thread 0 now starts running again, and stores into its per-thread `rcu_reader_gp` variable. The value it stores is `RCU_GP_CTR_BOTTOM_BIT+1` greater than that of the global `rcu_gp_ctr`.
4. Thread 0 acquires a reference to RCU-protected data element A.
5. Thread 1 now removes the data element A that thread 0 just acquired a reference to.
6. Thread 1 invokes `synchronize_rcu()`, which increments the global `rcu_gp_ctr` by `RCU_GP_CTR_BOTTOM_BIT`. It then checks all of the per-thread `rcu_reader_gp` variables, but thread 0's value (incorrectly) indicates that it started after thread 1's call to `synchronize_rcu()`, so thread 1 does not wait for thread 0 to complete its RCU read-side critical section.

7. Thread 1 then frees up data element A, which thread 0 is still referencing.

Note that scenario can also occur in the implementation presented in Appendix B.7.

One strategy for fixing this problem is to use 64-bit counters so that the time required to overflow them would exceed the useful lifetime of the computer system. Note that non-antique members of the 32-bit x86 CPU family allow atomic manipulation of 64-bit counters via the `cmpxchg64b` instruction.

Another strategy is to limit the rate at which grace periods are permitted to occur in order to achieve a similar effect. For example, `synchronize_rcu()` could record the last time that it was invoked, and any subsequent invocation would then check this time and block as needed to force the desired spacing. For example, if the low-order four bits of the counter were reserved for nesting, and if grace periods were permitted to occur at most ten times per second, then it would take more than 300 days for the counter to overflow. However, this approach is not helpful if there is any possibility that the system will be fully loaded with CPU-bound high-priority real-time threads for the full 300 days. (A remote possibility, perhaps, but best to consider it ahead of time.)

A third approach is to administratively abolish real-time threads from the system in question. In this case, the preempted process will age up in priority, thus getting to run long before the counter had a chance to overflow. Of course, this approach is less than helpful for real-time applications.

A final approach would be for `rcu_read_lock()` to recheck the value of the global `rcu_gp_ctr` after storing to its per-thread `rcu_reader_gp` counter, retrying if the new value of the global `rcu_gp_ctr` is inappropriate. This works, but introduces non-deterministic execution time into `rcu_read_lock()`. On the other hand, if your application is being preempted long enough for the counter to overflow, you have no hope of deterministic execution time in any case! ❑

Quick Quiz B.22:
Doesn't the additional memory barrier shown on line 14 of Listing B.18 greatly increase the overhead of `rcu_quiescent_state`? ■

Answer:
Indeed it does! An application using this implementation of RCU should therefore invoke `rcu_quiescent_state` sparingly, instead using `rcu_read_lock()` and `rcu_read_unlock()` most of the time.

However, this memory barrier is absolutely required so that other threads will see the store on lines 12–13 before any subsequent RCU read-side critical sections executed by the caller. ❑

Quick Quiz B.23:
Why are the two memory barriers on lines 11 and 14 of Listing B.18 needed? ■

Answer:
The memory barrier on line 11 prevents any RCU read-side critical sections that might precede the call to `rcu_thread_offline()` won't be reordered by either the compiler or the CPU to follow the assignment on lines 12–13. The memory barrier on line 14 is, strictly speaking, unnecessary, as it is illegal to have any RCU read-side critical sections following the call to `rcu_thread_offline()`. ❑

Quick Quiz B.24:
To be sure, the clock frequencies of POWER systems in 2008 were quite high, but even a 5 GHz clock frequency is insufficient to allow loops to be executed in 50 picoseconds! What is going on here? ■

Answer:
Since the measurement loop contains a pair of empty functions, the compiler optimizes it away. The measurement loop takes 1,000 passes between each call to `rcu_quiescent_state()`, so this measurement is roughly one thousandth of the overhead of a single call to `rcu_quiescent_state()`. ❑

Quick Quiz B.25:
Why would the fact that the code is in a library make any difference for how easy it is to use the RCU implementation shown in Listings B.18 and B.19? ■

Answer:
A library function has absolutely no control over the caller, and thus cannot force the caller to invoke `rcu_quiescent_state()` periodically. On the other hand, a library function that made many references to a given RCU-protected data structure might be able to invoke `rcu_thread_online()` upon entry, `rcu_quiescent_state()` periodically, and `rcu_thread_offline()` upon exit. ❑

www.dbooks.org

Quick Quiz B.26:
But what if you hold a lock across a call to `synchronize_rcu()`, and then acquire that same lock within an RCU read-side critical section? This should be a deadlock, but how can a primitive that generates absolutely no code possibly participate in a deadlock cycle? ■

Answer:
Please note that the RCU read-side critical section is in effect extended beyond the enclosing `rcu_read_lock()` and `rcu_read_unlock()`, out to the previous and next call to `rcu_quiescent_state()`. This `rcu_quiescent_state` can be thought of as an `rcu_read_unlock()` immediately followed by an `rcu_read_lock()`.

Even so, the actual deadlock itself will involve the lock acquisition in the RCU read-side critical section and the `synchronize_rcu()`, never the `rcu_quiescent_state()`. ❑

Quick Quiz B.27:
Given that grace periods are prohibited within RCU read-side critical sections, how can an RCU data structure possibly be updated while in an RCU read-side critical section? ■

Answer:
This situation is one reason for the existence of asynchronous grace-period primitives such as `call_rcu()`. This primitive may be invoked within an RCU read-side critical section, and the specified RCU callback will in turn be invoked at a later time, after a grace period has elapsed.

The ability to perform an RCU update while within an RCU read-side critical section can be extremely convenient, and is analogous to a (mythical) unconditional read-to-write upgrade for reader-writer locking. ❑

E.20 Why Memory Barriers?

Quick Quiz C.1:
Where does a writeback message originate from and where does it go to? ■

Answer:
The writeback message originates from a given CPU, or in some designs from a given level of a given CPU's cache—or even from a cache that might be shared among several CPUs. The key point is that a given cache does not have room for a given data item, so some other piece of data must be ejected from the cache to make room. If there is some other piece of data that is duplicated in some other cache or in memory, then that piece of data may be simply discarded, with no writeback message required.

On the other hand, if every piece of data that might be ejected has been modified so that the only up-to-date copy is in this cache, then one of those data items must be copied somewhere else. This copy operation is undertaken using a "writeback message".

The destination of the writeback message has to be something that is able to store the new value. This might be main memory, but it also might be some other cache. If it is a cache, it is normally a higher-level cache for the same CPU, for example, a level-1 cache might write back to a level-2 cache. However, some hardware designs permit cross-CPU writebacks, so that CPU 0's cache might send a writeback message to CPU 1. This would normally be done if CPU 1 had somehow indicated an interest in the data, for example, by having recently issued a read request.

In short, a writeback message is sent from some part of the system that is short of space, and is received by some other part of the system that can accommodate the data. ❑

Quick Quiz C.2:
What happens if two CPUs attempt to invalidate the same cache line concurrently? ■

Answer:
One of the CPUs gains access to the shared bus first, and that CPU "wins". The other CPU must invalidate its copy of the cache line and transmit an "invalidate acknowledge" message to the other CPU.

Of course, the losing CPU can be expected to immediately issue a "read invalidate" transaction, so the winning CPU's victory will be quite ephemeral. ❑

Quick Quiz C.3:
When an "invalidate" message appears in a large multiprocessor, every CPU must give an "invalidate acknowledge" response. Wouldn't the resulting "storm" of "invalidate acknowledge" responses totally saturate the system bus? ■

Answer:
It might, if large-scale multiprocessors were in fact implemented that way. Larger multiprocessors, particularly

NUMA machines, tend to use so-called "directory-based" cache-coherence protocols to avoid this and other problems. ❑

Quick Quiz C.4:
If SMP machines are really using message passing anyway, why bother with SMP at all? ■

Answer:
There has been quite a bit of controversy on this topic over the past few decades. One answer is that the cache-coherence protocols are quite simple, and therefore can be implemented directly in hardware, gaining bandwidths and latencies unattainable by software message passing. Another answer is that the real truth is to be found in economics due to the relative prices of large SMP machines and that of clusters of smaller SMP machines. A third answer is that the SMP programming model is easier to use than that of distributed systems, but a rebuttal might note the appearance of HPC clusters and MPI. And so the argument continues. ❑

Quick Quiz C.5:
How does the hardware handle the delayed transitions described above? ■

Answer:
Usually by adding additional states, though these additional states need not be actually stored with the cache line, due to the fact that only a few lines at a time will be transitioning. The need to delay transitions is but one issue that results in real-world cache coherence protocols being much more complex than the over-simplified MESI protocol described in this appendix. Hennessy and Patterson's classic introduction to computer architecture [HP95] covers many of these issues. ❑

Quick Quiz C.6:
What sequence of operations would put the CPUs' caches all back into the "invalid" state? ■

Answer:
There is no such sequence, at least in absence of special "flush my cache" instructions in the CPU's instruction set. Most CPUs do have such instructions. ❑

Quick Quiz C.7:
But if the main purpose of store buffers is to hide acknowledgment latencies in multiprocessor cache-coherence protocols, why do uniprocessors also have store buffers? ■

Answer:
Because the purpose of store buffers is not just to hide acknowledgement latencies in multiprocessor cache-coherence protocols, but to hide memory latencies in general. Because memory is much slower than is cache on uniprocessors, store buffers on uniprocessors can help to hide write-miss latencies. ❑

Quick Quiz C.8:
In step 1 above, why does CPU 0 need to issue a "read invalidate" rather than a simple "invalidate"? ■

Answer:
Because the cache line in question contains more than just the variable `a`. ❑

Quick Quiz C.9:
After step 15 in Appendix C.3.3 on page 415, both CPUs might drop the cache line containing the new value of "b". Wouldn't that cause this new value to be lost? ■

Answer:
It might, and that is why real hardware takes steps to avoid this problem. A traditional approach, pointed out by Vasilevsky Alexander, is to write this cache line back to main memory before marking the cache line as "shared". A more efficient (though more complex) approach is to use additional state to indicate whether or not the cache line is "dirty", allowing the writeback to happen. Year-2000 systems went further, using much more state in order to avoid redundant writebacks [CSG99, Figure 8.42]. It would be reasonable to assume that complexity has not decreased in the meantime. ❑

Quick Quiz C.10:
In step 1 of the first scenario in Appendix C.4.3, why is an "invalidate" sent instead of a "read invalidate" message? Doesn't CPU 0 need the values of the other variables that share this cache line with "a"? ■

Answer:
CPU 0 already has the values of these variables, given that it has a read-only copy of the cache line containing "a". Therefore, all CPU 0 need do is to cause the other CPUs

www.dbooks.org

to discard their copies of this cache line. An "invalidate" message therefore suffices. ❑

Quick Quiz C.11:
Say what??? Why do we need a memory barrier here, given that the CPU cannot possibly execute the `assert()` until after the `while` loop completes? ■

Answer:
Suppose that memory barrier was omitted.

Keep in mind that CPUs are free to speculatively execute later loads, which can have the effect of executing the assertion before the `while` loop completes. Furthermore, compilers assume that only the currently executing thread is updating the variables, and this assumption allows the compiler to hoist the load of `a` to precede the loop.

In fact, some compilers would transform the loop to a branch around an infinite loop as follows:

```
1  void foo(void)
2  {
3    a = 1;
4    smp_mb();
5    b = 1;
6  }
7
8  void bar(void)
9  {
10   if (b == 0)
11     for (;;)
12       continue;
13   assert(a == 1);
14 }
```

Given this optimization, the code would behave in a completely different way than the original code. If `bar()` observed "`b == 0`", the assertion could of course not be reached at all due to the infinite loop. However, if `bar()` loaded the value "1" just as "`foo()`" stored it, the CPU might still have the old zero value of "`a`" in its cache, which would cause the assertion to fire. You should of course use volatile casts (for example, those volatile casts implied by the C11 relaxed atomic load operation) to prevent the compiler from optimizing your parallel code into oblivion. But volatile casts would not prevent a weakly ordered CPU from loading the old value for "`a`" from its cache, which means that this code also requires the explicit memory barrier in "`bar()`".

In short, both compilers and CPUs aggressively apply code-reordering optimizations, so you must clearly communicate your constraints using the compiler directives and memory barriers provided for this purpose. ❑

Quick Quiz C.12:
Does the guarantee that each CPU sees its own memory accesses in order also guarantee that each user-level thread will see its own memory accesses in order? Why or why not? ■

Answer:
No. Consider the case where a thread migrates from one CPU to another, and where the destination CPU perceives the source CPU's recent memory operations out of order. To preserve user-mode sanity, kernel hackers must use memory barriers in the context-switch path. However, the locking already required to safely do a context switch should automatically provide the memory barriers needed to cause the user-level task to see its own accesses in order. That said, if you are designing a super-optimized scheduler, either in the kernel or at user level, please keep this scenario in mind! ❑

Quick Quiz C.13:
Could this code be fixed by inserting a memory barrier between CPU 1's "while" and assignment to "c"? Why or why not? ■

Answer:
No. Such a memory barrier would only force ordering local to CPU 1. It would have no effect on the relative ordering of CPU 0's and CPU 1's accesses, so the assertion could still fail. However, all mainstream computer systems provide one mechanism or another to provide "transitivity", which provides intuitive causal ordering: if B saw the effects of A's accesses, and C saw the effects of B's accesses, then C must also see the effects of A's accesses. In short, hardware designers have taken at least a little pity on software developers. ❑

Quick Quiz C.14:
Suppose that lines 3–5 for CPUs 1 and 2 in Listing C.3 are in an interrupt handler, and that the CPU 2's line 9 runs at process level. In other words, the code in all three columns of the table runs on the same CPU, but the first two columns run in an interrupt handler, and the third column runs at process level, so that the code in third column can be interrupted by the code in the first two columns. What changes, if any, are required to enable the code to work correctly, in other words, to prevent the assertion from firing? ■

Answer:
The assertion must ensure that the load of "e" precedes

that of "a". In the Linux kernel, the `barrier()` primitive may be used to accomplish this in much the same way that the memory barrier was used in the assertions in the previous examples. For example, the assertion can be modified as follows:

```
r1 = e;
barrier();
assert(r1 == 0 || a == 1);
```

No changes are needed to the code in the first two columns, because interrupt handlers run atomically from the perspective of the interrupted code. ❑

Quick Quiz C.15:
If CPU 2 executed an `assert(e==0||c==1)` in the example in Listing C.3, would this assert ever trigger? ■

Answer:
The result depends on whether the CPU supports "transitivity". In other words, CPU 0 stored to "e" after seeing CPU 1's store to "c", with a memory barrier between CPU 0's load from "c" and store to "e". If some other CPU sees CPU 0's store to "e", is it also guaranteed to see CPU 1's store?

All CPUs I am aware of claim to provide transitivity. ❑

www.dbooks.org

Dictionaries are inherently circular in nature.

"Self Reference in word definitions",
David Levary et al.

Glossary

Associativity: The number of cache lines that can be held simultaneously in a given cache, when all of these cache lines hash identically in that cache. A cache that could hold four cache lines for each possible hash value would be termed a "four-way set-associative" cache, while a cache that could hold only one cache line for each possible hash value would be termed a "direct-mapped" cache. A cache whose associativity was equal to its capacity would be termed a "fully associative" cache. Fully associative caches have the advantage of eliminating associativity misses, but, due to hardware limitations, fully associative caches are normally quite limited in size. The associativity of the large caches found on modern microprocessors typically range from two-way to eight-way.

Associativity Miss: A cache miss incurred because the corresponding CPU has recently accessed more data hashing to a given set of the cache than will fit in that set. Fully associative caches are not subject to associativity misses (or, equivalently, in fully associative caches, associativity and capacity misses are identical).

Atomic: An operation is considered "atomic" if it is not possible to observe any intermediate state. For example, on most CPUs, a store to a properly aligned pointer is atomic, because other CPUs will see either the old value or the new value, but are guaranteed not to see some mixed value containing some pieces of the new and old values.

Atomic Read-Modify-Write Operation: An atomic operation that both reads and writes memory is considered an atomic read-modify-write operation, or atomic RMW operation for short. Although the value written usually depends on the value read, `atomic_xchg()` is the exception that proves this rule.

Bounded Wait Free: A forward-progress guarantee in which every thread makes progress within a specific finite period of time, the specific time being the bound.

Cache: In modern computer systems, CPUs have caches in which to hold frequently used data. These caches can be thought of as hardware hash tables with very simple hash functions, but in which each hash bucket (termed a "set" by hardware types) can hold only a limited number of data items. The number of data items that can be held by each of a cache's hash buckets is termed the cache's "associativity". These data items are normally called "cache lines", which can be thought of a fixed-length blocks of data that circulate among the CPUs and memory.

Cache Coherence: A property of most modern SMP machines where all CPUs will observe a sequence of values for a given variable that is consistent with at least one global order of values for that variable. Cache coherence also guarantees that at the end of a group of stores to a given variable, all CPUs will agree on the final value for that variable. Note that cache coherence applies only to the series of values taken on by a single variable. In contrast, the memory consistency model for a given machine describes the order in which loads and stores to groups of variables will appear to occur. See Section 15.2.6 for more information.

Cache Coherence Protocol: A communications protocol, normally implemented in hardware, that enforces memory consistency and ordering, preventing different CPUs from seeing inconsistent views of data held in their caches.

Cache Geometry: The size and associativity of a cache is termed its geometry. Each cache may be thought of as a two-dimensional array, with rows of cache lines ("sets") that have the same hash value, and columns of cache lines ("ways") in which every cache line has a different hash value. The associativity of a given cache is its number of columns (hence the

www.dbooks.org

name "way"—a two-way set-associative cache has two "ways"), and the size of the cache is its number of rows multiplied by its number of columns.

Cache Line: (1) The unit of data that circulates among the CPUs and memory, usually a moderate power of two in size. Typical cache-line sizes range from 16 to 256 bytes.
(2) A physical location in a CPU cache capable of holding one cache-line unit of data.
(3) A physical location in memory capable of holding one cache-line unit of data, but that it also aligned on a cache-line boundary. For example, the address of the first word of a cache line in memory will end in 0x00 on systems with 256-byte cache lines.

Cache Miss: A cache miss occurs when data needed by the CPU is not in that CPU's cache. The data might be missing because of a number of reasons, including: (1) this CPU has never accessed the data before ("startup" or "warmup" miss), (2) this CPU has recently accessed more data than would fit in its cache, so that some of the older data had to be removed ("capacity" miss), (3) this CPU has recently accessed more data in a given set[1] than that set could hold ("associativity" miss), (4) some other CPU has written to the data (or some other data in the same cache line) since this CPU has accessed it ("communication miss"), or (5) this CPU attempted to write to a cache line that is currently read-only, possibly due to that line being replicated in other CPUs' caches.

Capacity Miss: A cache miss incurred because the corresponding CPU has recently accessed more data than will fit into the cache.

Clash Free: A forward-progress guarantee in which, in the absence of contention, at least one thread makes progress within a finite period of time.

Code Locking: A simple locking design in which a "global lock" is used to protect a set of critical sections, so that access by a given thread to that set is granted or denied based only on the set of threads currently occupying the set of critical sections, not based on what data the thread intends to access. The scalability of a code-locked program is limited by the code; increasing the size of the data set will normally not increase scalability (in fact, will typically *decrease* scalability by increasing "lock contention"). Contrast with "data locking".

Communication Miss: A cache miss incurred because some other CPU has written to the cache line since the last time this CPU accessed it.

Critical Section: A section of code guarded by some synchronization mechanism, so that its execution constrained by that primitive. For example, if a set of critical sections are guarded by the same global lock, then only one of those critical sections may be executing at a given time. If a thread is executing in one such critical section, any other threads must wait until the first thread completes before executing any of the critical sections in the set.

Data Locking: A scalable locking design in which each instance of a given data structure has its own lock. If each thread is using a different instance of the data structure, then all of the threads may be executing in the set of critical sections simultaneously. Data locking has the advantage of automatically scaling to increasing numbers of CPUs as the number of instances of data grows. Contrast with "code locking".

Deadlock Free: A forward-progress guarantee in which, in the absence of failures, at least one thread makes progress within a finite period of time.

Direct-Mapped Cache: A cache with only one way, so that it may hold only one cache line with a given hash value.

Efficiency: A measure of effectiveness normally expressed as a ratio of of some metric actually achieved to some maximum value. The maximum value might be a theoretical maximum, but in parallel programming is often based on the corresponding measured single-threaded metric.

Embarrassingly Parallel: A problem or algorithm where adding threads does not significantly increase the overall cost of the computation, resulting in linear speedups as threads are added (assuming sufficient CPUs are available).

Exclusive Lock: An exclusive lock is a mutual-exclusion mechanism that permits only one thread at a time into the set of critical sections guarded by that lock.

[1] In hardware-cache terminology, the word "set" is used in the same way that the word "bucket" is used when discussing software caches.

False Sharing: If two CPUs each frequently write to one of a pair of data items, but the pair of data items are located in the same cache line, this cache line will be repeatedly invalidated, "ping-ponging" back and forth between the two CPUs' caches. This is a common cause of "cache thrashing", also called "cacheline bouncing" (the latter most commonly in the Linux community). False sharing can dramatically reduce both performance and scalability.

Fragmentation: A memory pool that has a large amount of unused memory, but not laid out to permit satisfying a relatively small request is said to be fragmented. External fragmentation occurs when the space is divided up into small fragments lying between allocated blocks of memory, while internal fragmentation occurs when specific requests or types of requests have been allotted more memory than they actually requested.

Fully Associative Cache: A fully associative cache contains only one set, so that it can hold any subset of memory that fits within its capacity.

Grace Period: A grace period is any contiguous time interval such that any RCU read-side critical section that began before the start of that interval has completed before the end of that same interval. Many RCU implementations define a grace period to be a time interval during which each thread has passed through at least one quiescent state. Since RCU read-side critical sections by definition cannot contain quiescent states, these two definitions are almost always interchangeable.

Hazard Pointer: A scalable counterpart to a reference counter in which an object's reference count is represented implicitly by a count of the number of special hazard pointers referencing that object.

Heisenbug: A timing-sensitive bug that disappears from sight when you add print statements or tracing in an attempt to track it down.

Hot Spot: Data structure that is very heavily used, resulting in high levels of contention on the corresponding lock. One example of this situation would be a hash table with a poorly chosen hash function.

Humiliatingly Parallel: A problem or algorithm where adding threads significantly *decreases* the overall cost of the computation, resulting in large superlinear speedups as threads are added (assuming sufficient CPUs are available).

Invalidation: When a CPU wishes to write to a data item, it must first ensure that this data item is not present in any other CPUs' cache. If necessary, the item is removed from the other CPUs' caches via "invalidation" messages from the writing CPUs to any CPUs having a copy in their caches.

IPI: Inter-processor interrupt, which is an interrupt sent from one CPU to another. IPIs are used heavily in the Linux kernel, for example, within the scheduler to alert CPUs that a high-priority process is now runnable.

IRQ: Interrupt request, often used as an abbreviation for "interrupt" within the Linux kernel community, as in "irq handler".

Latency: The wall-clock time required for a given operation to complete.

Linearizable: A sequence of operations is "linearizable" if there is at least one global ordering of the sequence that is consistent with the observations of all CPUs and/or threads. Linearizability is much prized by many researchers, but less useful in practice than one might expect [HKLP12].

Lock: A software abstraction that can be used to guard critical sections, as such, an example of a "mutual exclusion mechanism". An "exclusive lock" permits only one thread at a time into the set of critical sections guarded by that lock, while a "reader-writer lock" permits any number of reading threads, or but one writing thread, into the set of critical sections guarded by that lock. (Just to be clear, the presence of a writer thread in any of a given reader-writer lock's critical sections will prevent any reader from entering any of that lock's critical sections and vice versa.)

Lock Contention: A lock is said to be suffering contention when it is being used so heavily that there is often a CPU waiting on it. Reducing lock contention is often a concern when designing parallel algorithms and when implementing parallel programs.

Lock Free: A forward-progress guarantee in which at least one thread makes progress within a finite period of time.

www.dbooks.org

Memory: From the viewpoint of memory models, the main memory, caches, and store buffers in which values might be stored. However, this term is often used to denote the main memory itself, excluding caches and store buffers.

Memory Consistency: A set of properties that impose constraints on the order in which accesses to groups of variables appear to occur. Memory consistency models range from sequential consistency, a very constraining model popular in academic circles, through process consistency, release consistency, and weak consistency.

MESI Protocol: The cache-coherence protocol featuring modified, exclusive, shared, and invalid (MESI) states, so that this protocol is named after the states that the cache lines in a given cache can take on. A modified line has been recently written to by this CPU, and is the sole representative of the current value of the corresponding memory location. An exclusive cache line has not been written to, but this CPU has the right to write to it at any time, as the line is guaranteed not to be replicated into any other CPU's cache (though the corresponding location in main memory is up to date). A shared cache line is (or might be) replicated in some other CPUs' cache, meaning that this CPU must interact with those other CPUs before writing to this cache line. An invalid cache line contains no value, instead representing "empty space" in the cache into which data from memory might be loaded.

Mutual-Exclusion Mechanism: A software abstraction that regulates threads' access to "critical sections" and corresponding data.

NMI: Non-maskable interrupt. As the name indicates, this is an extremely high-priority interrupt that cannot be masked. These are used for hardware-specific purposes such as profiling. The advantage of using NMIs for profiling is that it allows you to profile code that runs with interrupts disabled.

NUCA: Non-uniform cache architecture, where groups of CPUs share caches and/or store buffers. CPUs in a group can therefore exchange cache lines with each other much more quickly than they can with CPUs in other groups. Systems comprised of CPUs with hardware threads will generally have a NUCA architecture.

NUMA: Non-uniform memory architecture, where memory is split into banks and each such bank is "close" to a group of CPUs, the group being termed a "NUMA node". An example NUMA machine is Sequent's NUMA-Q system, where each group of four CPUs had a bank of memory nearby. The CPUs in a given group can access their memory much more quickly than another group's memory.

NUMA Node: A group of closely placed CPUs and associated memory within a larger NUMA machines.

Obstruction Free: A forward-progress guarantee in which, in the absence of contention, every thread makes progress within a finite period of time.

Overhead: Operations that must be executed, but which do not contribute directly to the work that must be accomplished. For example, lock acquisition and release is normally considered to be overhead, and specifically to be synchronization overhead.

Performance: Rate at which work is done, expressed as work per unit time. If this work is fully serialized, then the performance will be the reciprocal of the mean latency of the work items.

Pipelined CPU: A CPU with a pipeline, which is an internal flow of instructions internal to the CPU that is in some way similar to an assembly line, with many of the same advantages and disadvantages. In the 1960s through the early 1980s, pipelined CPUs were the province of supercomputers, but started appearing in microprocessors (such as the 80486) in the late 1980s.

Process Consistency: A memory-consistency model in which each CPU's stores appear to occur in program order, but in which different CPUs might see accesses from more than one CPU as occurring in different orders.

Program Order: The order in which a given thread's instructions would be executed by a now-mythical "in-order" CPU that completely executed each instruction before proceeding to the next instruction. (The reason such CPUs are now the stuff of ancient myths and legends is that they were extremely slow. These dinosaurs were one of the many victims of Moore's-Law-driven increases in CPU clock frequency. Some claim that these beasts will roam the earth once again, others vehemently disagree.)

Quiescent State: In RCU, a point in the code where there can be no references held to RCU-protected data structures, which is normally any point outside of an RCU read-side critical section. Any interval of time during which all threads pass through at least one quiescent state each is termed a "grace period".

Read-Copy Update (RCU): A synchronization mechanism that can be thought of as a replacement for reader-writer locking or reference counting. RCU provides extremely low-overhead access for readers, while writers incur additional overhead maintaining old versions for the benefit of pre-existing readers. Readers neither block nor spin, and thus cannot participate in deadlocks, however, they also can see stale data and can run concurrently with updates. RCU is thus best-suited for read-mostly situations where stale data can either be tolerated (as in routing tables) or avoided (as in the Linux kernel's System V IPC implementation).

Read-Side Critical Section: A section of code guarded by read-acquisition of some reader-writer synchronization mechanism. For example, if one set of critical sections are guarded by read-acquisition of a given global reader-writer lock, while a second set of critical section are guarded by write-acquisition of that same reader-writer lock, then the first set of critical sections will be the read-side critical sections for that lock. Any number of threads may concurrently execute the read-side critical sections, but only if no thread is executing one of the write-side critical sections.

Reader-Writer Lock: A reader-writer lock is a mutual-exclusion mechanism that permits any number of reading threads, or but one writing thread, into the set of critical sections guarded by that lock. Threads attempting to write must wait until all pre-existing reading threads release the lock, and, similarly, if there is a pre-existing writer, any threads attempting to write must wait for the writer to release the lock. A key concern for reader-writer locks is "fairness": can an unending stream of readers starve a writer or vice versa.

Real Time: A situation in which getting the correct result is not sufficient, but where this result must also be obtained within a given amount of time.

Scalability: A measure of how effectively a given system is able to utilize additional resources. For parallel computing, the additional resources are usually additional CPUs.

Sequence Lock: A reader-writer synchronization mechanism in which readers retry their operations if a writer was present.

Sequential Consistency: A memory-consistency model where all memory references appear to occur in an order consistent with a single global order, and where each CPU's memory references appear to all CPUs to occur in program order.

Starvation Free: A forward-progress guarantee in which, in the absence of failures, every thread makes progress within a finite period of time.

Store Buffer: A small set of internal registers used by a given CPU to record pending stores while the corresponding cache lines are making their way to that CPU. Also called "store queue".

Store Forwarding: An arrangement where a given CPU refers to its store buffer as well as its cache so as to ensure that the software sees the memory operations performed by this CPU as if they were carried out in program order.

Superscalar CPU: A scalar (non-vector) CPU capable of executing multiple instructions concurrently. This is a step up from a pipelined CPU that executes multiple instructions in an assembly-line fashion—in a superscalar CPU, each stage of the pipeline would be capable of handling more than one instruction. For example, if the conditions were exactly right, the Intel Pentium Pro CPU from the mid-1990s could execute two (and sometimes three) instructions per clock cycle. Thus, a 200 MHz Pentium Pro CPU could "retire", or complete the execution of, up to 400 million instructions per second.

Synchronization: Means for avoiding destructive interactions among CPUs or threads. Synchronization mechanisms include atomic RMW operations, memory barriers, locking, reference counting, hazard pointers, sequence locking, RCU, non-blocking synchronization, and transactional memory.

Teachable: A topic, concept, method, or mechanism that teachers believe that they understand completely and are therefore comfortable teaching.

www.dbooks.org

Throughput: A performance metric featuring work items completed per unit time.

Transactional Lock Elision (TLE): The use of transactional memory to emulate locking. Synchronization is instead carried out by conflicting accesses to the data to be protected by the lock. In some cases, this can increase performance because TLE avoids contention on the lock word [PD11, Kle14, FIMR16, PMDY20].

Transactional Memory (TM): A synchronization mechanism that gathers groups of memory accesses so as to execute them atomically from the viewpoint of transactions on other CPUs or threads.

Unteachable: A topic, concept, method, or mechanism that the teacher does not understand well is therefore uncomfortable teaching.

Vector CPU: A CPU that can apply a single instruction to multiple items of data concurrently. In the 1960s through the 1980s, only supercomputers had vector capabilities, but the advent of MMX in x86 CPUs and VMX in PowerPC CPUs brought vector processing to the masses.

Wait Free: A forward-progress guarantee in which every thread makes progress within a finite period of time.

Write Miss: A cache miss incurred because the corresponding CPU attempted to write to a cache line that is read-only, most likely due to its being replicated in other CPUs' caches.

Write-Side Critical Section: A section of code guarded by write-acquisition of some reader-writer synchronization mechanism. For example, if one set of critical sections are guarded by write-acquisition of a given global reader-writer lock, while a second set of critical section are guarded by read-acquisition of that same reader-writer lock, then the first set of critical sections will be the write-side critical sections for that lock. Only one thread may execute in the write-side critical section at a time, and even then only if there are no threads are executing concurrently in any of the corresponding read-side critical sections.

www.dbooks.org

Bibliography

[AA14] Maya Arbel and Hagit Attiya. Concurrent updates with RCU: Search tree as an example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, page 196âĂŞ205, Paris, France, 2014. ACM.

[AAKL06] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, and Charles E. Leiserson. Unbounded transactional memory. *IEEE Micro*, pages 59–69, January-February 2006.

[AB13] Samy Al Bahra. Nonblocking algorithms and scalable multicore programming. *Commun. ACM*, 56(7):50–61, July 2013.

[ABD+97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 1–14, New York, NY, October 1997.

[ACA+18] A. Aljuhni, C. E. Chow, A. Aljaedi, S. Yusuf, and F. Torres-Reyes. Towards understanding application performance and system behavior with the full dynticks feature. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 394–401, 2018.

[ACHS13] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free?, December 2013. ArXiv:1311.3200v2.

[ACMS03] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, pages 297–310, San Antonio, Texas, USA, June 2003. USENIX Association.

[Ada11] Andrew Adamatzky. Slime mould solves maze in one pass . . . assisted by gradient of chemo-attractants, August 2011. arXiv:1108.4956.

[ADF+19] Jade Alglave, Will Deacon, Boqun Feng, David Howells, Daniel Lustig, Luc Maranget, Paul E. McKenney, Andrea Parri, Nicholas Piggin, Alan Stern, Akira Yokosawa, and Peter Zijlstra. Who's afraid of a big bad optimizing compiler?, July 2019. Linux Weekly News.

[Adv02] Advanced Micro Devices. *AMD x86-64 Architecture Programmer's Manual Volumes 1–5*, 2002.

[AGH+11a] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *38th ACM SIGACT-SIGPLAN*

Edition.2-rc11

www.dbooks.org

Symposium on Principles of Programming Languages, pages 487–498, Austin, TX, USA, 2011. ACM.

[AGH+11b] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. *SIGPLAN Not.*, 46(1):487–498, January 2011.

[AHM09] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 69–78, Calgary, AB, Canada, 2009. ACM.

[AHS+03] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, January 2003.

[AKK+14] Dan Alistarh, Justin Kopinsky, Petr Kuznetsov, Srivatsan Ravi, and Nir Shavit. Inherent limitations of hybrid transactional memory. *CoRR*, abs/1405.5689, 2014.

[AKNT13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *Proceedings of the 22nd European conference on Programming Languages and Systems*, ESOP'13, pages 512–532, Rome, Italy, 2013. Springer-Verlag.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient Bounded Model Checking of concurrent software. In *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.

[Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[Alg13] Jade Alglave. Weakness is a virtue. In *(EC)^2 2013: 6th International Workshop on Exploiting Concurrency Efficiently and Correctly*, page 3, 2013.

[AM15] Maya Arbel and Adam Morrison. Predicate RCU: An RCU for scalable concurrent updates. *SIGPLAN Not.*, 50(8):21–30, January 2015.

[Amd67] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, AFIPS '67 (Spring), pages 483–485, Atlantic City, New Jersey, 1967. Association for Computing Machinery.

[AMD20] AMD. Professional compute products - GPUOpen, March 2020. `https://gpuopen.com/professional-compute/`.

[AMM+17a] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. A formal kernel memory-ordering model (part 1), April 2017. `https://lwn.net/Articles/718628/`.

[AMM+17b] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. A formal kernel memory-ordering model (part 2), April 2017. `https://lwn.net/Articles/720550/`.

[AMM+18] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 405–418, Williamsburg, VA, USA, 2018. ACM.

[AMP+11] Jade Alglave, Luc Maranget, Pankaj Pawan, Susmit Sarkar, Peter Sewell, Derek Williams, and Francesco Zappa Nardelli. PPCMEM/ARMMEM: A tool for exploring the POWER and ARM memory models, June 2011. `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/pldi105-sarkar.pdf`.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data-mining for weak memory. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 40–40, Edinburgh, United Kingdom, 2014. ACM.

[And90] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[And91] Gregory R. Andrews. *Concurrent Programming, Principles, and Practices*. Benjamin Cummins, 1991.

[And19] Jim Anderson. Software transactional memory for real-time systems, August 2019. `https://www.cs.unc.edu/~anderson/projects/rtstm.html`.

[ARM10] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*, 2010.

[ARM17] ARM Limited. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*, 2017.

[Ash15] Mike Ash. Concurrent memory deallocation in the objective-c runtime, May 2015. mikeash.com: just this guy, you know?

[ATC+11] Ege Akpinar, Sasa Tomic, Adrian Cristal, Osman Unsal, and Mateo Valero. A comprehensive study of conflict resolution policies in hardware transactional memory. In *TRANSACT 2011*, New Orleans, LA, USA, June 2011. ACM SIGPLAN.

[ATS09] Ali-Reza Adl-Tabatabai and Tatiana Shpeisman. Draft specification of transactional language constructs for C++, August 2009. URL: `https://software.intel.com/sites/default/files/ee/47/21569` (may need to append .pdf to view after download).

[Att10] Hagit Attiya. The inherent complexity of transactional memory and what to do about it. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 1–5, Zurich, Switzerland, 2010. ACM.

[BA01] Jeff Bonwick and Jonathan Adams. Magazines and vmem: Extending the slab allocator to many CPUs and arbitrary resources. In *USENIX Annual Technical Conference, General Track 2001*, pages 15–33, 2001.

[Bah11a] Samy Al Bahra. ck_epoch: Support per-object destructors, October 2011. `https://github.com/concurrencykit/ck/commit/10ffb2e6f1737a30e2dcf3862d105ad45fcd60a4`.

www.dbooks.org

[Bah11b] Samy Al Bahra. ck_hp.c, February 2011. Hazard pointers: https://github.com/concurrencykit/ck/blob/master/src/ck_hp.c.

[Bah11c] Samy Al Bahra. ck_sequence.h, February 2011. Sequence locking: https://github.com/concurrencykit/ck/blob/master/include/ck_sequence.h.

[Bas18] JF Bastien. P1152R0: Deprecating volatile, October 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1152r0.html.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66âĂŞ75, February 2010.

[BCR03] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. *SIGPLAN Not.*, 38(1):285–298, 2003.

[BD13] Paolo Bonzini and Mike Day. RCU implementation for Qemu, August 2013. https://lists.gnu.org/archive/html/qemu-devel/2013-08/msg02055.html.

[BD14] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 7:1–7:6, Edinburgh, United Kingdom, 2014. ACM.

[Bec11] Pete Becker. Working draft, standard for programming language C++, February 2011. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

[BG87] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Inc., 1987.

[BGHZ16] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 349–359, Pacific Grove, California, USA, 2016. ACM.

[BGOS18] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. Racerd: Compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[BGV17] Hans-J. Boehm, Olivier Giroux, and Viktor Vafeiades. P0668r1: Revising the C++ memory model, July 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0668r1.html.

[Bha14] Srivatsa S. Bhat. percpu_rwlock: Implement the core design of per-CPU reader-writer locks, February 2014. https://patchwork.kernel.org/patch/2157401/.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, Chichester, West Sussex, England, 2007.

[Bir89] Andrew D. Birrell. *An Introduction to Programming with Threads*. Digital Systems Research Center, January 1989.

[BJ12] Rex Black and Capers Jones. Economics of software quality: An interview with Capers Jones, part 1 of 2 (podcast transcript), January 2012. https://www.informit.com/articles/article.aspx?p=1824791.

[BK85] Bob Beck and Bob Kasten. VLSI assist in building a multiprocessor UNIX system. In *USENIX Conference Proceedings*, pages 255–275, Portland, OR, June 1985. USENIX Association.

[BLM05] C. Blundell, E. C. Lewis, and M. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005. Available: http://acg.cis.upenn.edu/papers/wddd05_atomic_semantics.pdf [Viewed February 28, 2021].

[BLM06] C. Blundell, E. C. Lewis, and M. Martin. Subtleties of transactional memory and atomicity semantics. *Computer Architecture Letters*, 5(2), 2006. Available: http://acg.cis.upenn.edu/papers/cal06_atomic_semantics.pdf [Viewed February 28, 2021].

[BM18] JF Bastien and Paul E. McKenney. P0750r1: Consume, February 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0750r1.html.

[BMMM05] Luke Browning, Thomas Mathews, Paul E. McKenney, and James Moody. Apparatus, method, and computer program product for converting simple locks in a multiprocessor system. US Patent 6,842,809, Assigned to International Business Machines Corporation, Washington, DC, January 2005.

[BMN+15] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307. Springer Berlin Heidelberg, 2015.

[BMP08] R. F. Berry, P. E. McKenney, and F. N. Parr. Responsive systems: An introduction. *IBM Systems Journal*, 47(2):197–206, April 2008.

[Boe05] Hans-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6):261–268, June 2005.

[Boe09] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *HOTPAR 2009*, page 6, Berkeley, CA, USA, March 2009. Available: https://www.usenix.org/event/hotpar09/tech/full_papers/boehm/boehm.pdf [Viewed May 24, 2009].

[Boe20] Hans Boehm. "Undefined behavior" and the concurrency memory model, August 2020. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2215r0.pdf.

[Boh01] Kristoffer Bohmann. Response time still matters, July 2001. URL: http://www.bohmann.dk/articles/response_time_still_matters.html [broken, November 2016].

[Bon13] Paolo Bonzini. seqlock: introduce read-write seqlock, September 2013. https://git.qemu.org/?p=qemu.git;a=commit;h=ea753d81e8b085d679f13e4a6023e003e9854d51.

[Bon15] Paolo Bonzini. rcu: add rcu library, February 2015. https://git.qemu.org/?p=qemu.git;a=commit;h=7911747bd46123ef8d8eef2ee49422bb8a4b274f.

www.dbooks.org

[Bon21a] Paolo Bonzini. An introduction to lockless algorithms, February 2021. Available: https://lwn.net/Articles/844224/ [Viewed February 19, 2021].

[Bon21b] Paolo Bonzini. Lockless patterns: an introduction to compare-and-swap, March 2021. Available: https://lwn.net/Articles/847973/ [Viewed March 13, 2021].

[Bon21c] Paolo Bonzini. Lockless patterns: full memory barriers, March 2021. Available: https://lwn.net/Articles/847481/ [Viewed March 8, 2021].

[Bon21d] Paolo Bonzini. Lockless patterns: relaxed access and partial memory barriers, February 2021. Available: https://lwn.net/Articles/846700/ [Viewed February 27, 2021].

[Bor06] Richard Bornat. Dividing the sheep from the goats, January 2006. Seminar at School of Computing, Univ. of Kent. Abstract is available at https://www.cs.kent.ac.uk/seminar_archive/2005_06/abs_2006_01_24.html. Retracted in July 2014: http://www.eis.mdx.ac.uk/staffpages/r_bornat/papers/camel_hump_retraction.pdf.

[Bos10] Keith Bostic. Switch lockless programming style from epoch to hazard references, January 2010. https://github.com/wiredtiger/wiredtiger/commit/dddc21014fc494a956778360a14d96c762495e09.

[BPP+16] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, December 2016.

[Bra07] Reg Braithwaite. Don't overthink fizzbuzz, January 2007. http://weblog.raganwald.com/2007/01/dont-overthink-fizzbuzz.html.

[Bra11] Björn Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011. URL: https://www.cs.unc.edu/~anderson/diss/bbbdiss.pdf.

[Bro15a] Neil Brown. Pathname lookup in Linux, June 2015. https://lwn.net/Articles/649115/.

[Bro15b] Neil Brown. RCU-walk: faster pathname lookup in Linux, July 2015. https://lwn.net/Articles/649729/.

[Bro15c] Neil Brown. A walk among the symlinks, July 2015. https://lwn.net/Articles/650786/.

[BS75] Paul J. Brown and Ronald M. Smith. Shared data controlled by a plurality of users, May 1975. US Patent 3,886,525, filed June 29, 1973.

[BS14] Mark Batty and Peter Sewell. The thin-air problem, February 2014. https://www.cl.cam.ac.uk/~pes20/cpp/notes42.html.

[But97] David Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Boston, MA, USA, 1997.

[BW14] Silas Boyd-Wickizer. *Optimizing Communications Bottlenecks in Multiprocessor Operating Systems Kernels*. PhD thesis, Massachusetts Institute of Technology, 2014. https://pdos.csail.mit.edu/papers/sbw-phd-thesis.pdf.

[BWCM+10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *9th USENIX Symposium on Operating System Design and Implementation*, pages 1–16, Vancouver, BC, Canada, October 2010. USENIX.

[CAK+96] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *International Conference on Configurable Distributed Systems (ICCDS'96)*, pages 108–115, Annapolis, MD, May 1996.

[CBF13] UPC Consortium, Dan Bonachea, and Gary Funck. UPC language and library specifications, version 1.3. Technical report, UPC Consortium, November 2013.

[CBM+08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *ACM Queue*, September 2008.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKZ12] Austin Clements, Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, pages 199–210, London, UK, March 2012. ACM.

[CKZ+13] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, Farminton, Pennsylvania, 2013. ACM.

[Cli09] Cliff Click. And now some hardware transactional memory comments..., February 2009. URL: `http://www.cliffc.org/blog/2009/02/25/and-now-some-hardware-transactional-memory-comments/`.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT electrical engineering and computer science series. MIT Press, 2001.

[CnRR18] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6), November 2018.

[Com01] Compaq Computer Corporation. Shared memory, threads, interprocess communication, August 2001. Zipped archive: `wiz_2637.txt` in `https://www.digiater.nl/openvms/freeware/v70/ask_the_wizard/wizard.zip`.

[Coo18] Byron Cook. Formal reasoning about the security of amazon web services. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 38–47, Cham, 2018. Springer International Publishing.

www.dbooks.org

[Cor02] Compaq Computer Corporation. *Alpha Architecture Reference Manual*. Digital Press, fourth edition, 2002.

[Cor03] Jonathan Corbet. Driver porting: mutual exclusion with seqlocks, February 2003. https://lwn.net/Articles/22818/.

[Cor04a] Jonathan Corbet. Approaches to realtime Linux, October 2004. URL: https://lwn.net/Articles/106010/.

[Cor04b] Jonathan Corbet. Finding kernel problems automatically, June 2004. https://lwn.net/Articles/87538/.

[Cor04c] Jonathan Corbet. Realtime preemption, part 2, October 2004. URL: https://lwn.net/Articles/107269/.

[Cor06a] Jonathan Corbet. The kernel lock validator, May 2006. Available: https://lwn.net/Articles/185666/ [Viewed: March 26, 2010].

[Cor06b] Jonathan Corbet. Priority inheritance in the kernel, April 2006. Available: https://lwn.net/Articles/178253/ [Viewed June 29, 2009].

[Cor10a] Jonathan Corbet. Dcache scalability and RCU-walk, December 2010. Available: https://lwn.net/Articles/419811/ [Viewed May 29, 2017].

[Cor10b] Jonathan Corbet. sys_membarrier(), January 2010. https://lwn.net/Articles/369567/.

[Cor11] Jonathan Corbet. How to ruin linus's vacation, July 2011. Available: https://lwn.net/Articles/452117/ [Viewed May 29, 2017].

[Cor12] Jonathan Corbet. ACCESS_ONCE(), August 2012. https://lwn.net/Articles/508991/.

[Cor13] Jonathan Corbet. (Nearly) full tickless operation in 3.10, May 2013. https://lwn.net/Articles/549580/.

[Cor14a] Jonathan Corbet. ACCESS_ONCE() and compiler bugs, December 2014. https://lwn.net/Articles/624126/.

[Cor14b] Jonathan Corbet. MCS locks and qspinlocks, March 2014. https://lwn.net/Articles/590243/.

[Cor14c] Jonathan Corbet. Relativistic hash tables, part 1: Algorithms, September 2014. https://lwn.net/Articles/612021/.

[Cor14d] Jonathan Corbet. Relativistic hash tables, part 2: Implementation, September 2014. https://lwn.net/Articles/612100/.

[Cor16a] Jonathan Corbet. Finding race conditions with KCSAN, June 2016. https://lwn.net/Articles/691128/.

[Cor16b] Jonathan Corbet. Time to move to C11 atomics?, June 2016. https://lwn.net/Articles/691128/.

[Cor18] Jonathan Corbet. membarrier(2), October 2018. https://man7.org/linux/man-pages/man2/membarrier.2.html.

[Cra93] Travis Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report 93-02-02, University of Washington, Seattle, Washington, February 1993.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., third edition, 2005. URL: https://lwn.net/Kernel/LDD3/.

[CSG99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: a Hardware/Software Approach*. Morgan Kaufman, 1999.

[cut17] crates.io user ticki. conc v0.5.0: Hazard-pointer-based concurrent memory reclamation, August 2017. `https://crates.io/crates/conc`.

[Dat82] C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley Publishing Company, 1982.

[DBA09] Saeed Dehnadi, Richard Bornat, and Ray Adams. Meta-analysis of the effect of consistency on success in early learning of programming. In *PPIG 2009*, pages 1–13, University of Limerick, Ireland, June 2009. Psychology of Programming Interest Group.

[DCW+11] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '11, page 39âĂŞ52, Newport Beach, CA, USA, 2011. ACM.

[Dea18] Will Deacon. [PATCH 00/10] kernel/locking: qspinlock improvements, April 2018. `https://lkml.kernel.org/r/1522947547-24081-1-git-send-email-will.deacon@arm.com`.

[Dea19] Will Deacon. Re: [PATCH 1/1] Fix: trace sched switch start/stop racy updates, August 2019. `https://lore.kernel.org/lkml/20190821103200.kpufwtviqhpbuv2n@willie-the-truck/`.

[Den15] Peter Denning. Perspectives on OS foundations. In *SOSP History Day 2015*, SOSP '15, pages 3:1–3:46, Monterey, California, 2015. ACM.

[Dep06] Department of Computing and Information Systems, University of Melbourne. CSIRAC, 2006. `https://cis.unimelb.edu.au/about/csirac/`.

[Des09a] Mathieu Desnoyers. *Low-Impact Operating System Tracing*. PhD thesis, Ecole Polytechnique de Montréal, December 2009. Available: `https://lttng.org/files/thesis/desnoyers-dissertation-2009-12-v27.pdf` [Viewed February 27, 2021].

[Des09b] Mathieu Desnoyers. [RFC git tree] userspace RCU (urcu) for Linux, February 2009. `https://liburcu.org`.

[DFGG11] Aleksandar Dragovejic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, pages 70–77, April 2011.

[DFLO19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62âĂŞ70, July 2019.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[DHK12] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Satisfiability solvers are static analyzers. In *Static Analysis Symposium (SAS)*, volume 7460 of *LNCS*, pages 317–333. Springer, 2012.

www.dbooks.org

[DHL+08] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the adaptive transactional memory test platform. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, pages 1–10, Salt Lake City, UT, USA, February 2008.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, Sept 1965.

[Dij68] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971. Available: `https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF` [Viewed January 13, 2008].

[DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queuing algorithm. *SIGCOMM '89*, pages 1–12, 1989.

[DLM+10] Dave Dice, Yossi Lev, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Oleszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 325–334, Thira, Santorini, Greece, 2010. ACM.

[DLMN09] Dave Dice, Yossi Lev, Mark Moir, and Dan Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 157–168, Washington, DC, USA, March 2009.

[DMD13] Mathieu Desnoyers, Paul E. McKenney, and Michel R. Dagenais. Multi-core systems modeling for formal verification of parallel algorithms. *SIGOPS Oper. Syst. Rev.*, 47(2):51–65, July 2013.

[DMLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979.

[DMS+12] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23:375–382, 2012.

[dO18a] Daniel Bristot de Oliveira. Deadline scheduler part 2 – details and usage, January 2018. URL: `https://lwn.net/Articles/743946/`.

[dO18b] Daniel Bristot de Oliveira. Deadline scheduling part 1 – overview and theory, January 2018. URL: `https://lwn.net/Articles/743740/`.

[dOCdO19] Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Modeling the behavior of threads in the PREEMPT_RT Linux kernel using automata. *SIGBED Rev.*, 16(3):63âĂŞ68, November 2019.

[Dov90] Ken F. Dove. A high capacity TCP/IP in parallel STREAMS. In *UKUUG Conference Proceedings*, London, June 1990.

[Dow20] Travis Downs. Gathering intel on Intel AVX-512 transitions, January 2020. `https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html`.

[Dre11] Ulrich Drepper. Futexes are tricky. Technical Report FAT2011, Red Hat, Inc., Raleigh, NC, USA, November 2011.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proc. International Symposium on Distributed Computing*. Springer Verlag, 2006.

[Duf10a] Joe Duffy. A (brief) retrospective on transactional memory, January 2010. `http://joeduffyblog.com/2010/01/03/a-brief-retrospective-on-transactional-memory/`.

[Duf10b] Joe Duffy. More thoughts on transactional memory, May 2010. `http://joeduffyblog.com/2010/05/16/more-thoughts-on-transactional-memory/`.

[Dug10] Abhinav Duggal. Stopping data races using redflag. Master's thesis, Stony Brook University, 2010.

[Eas71] William B. Easton. Process synchronization without long-term interlock. In *Proceedings of the Third ACM Symposium on Operating Systems Principles*, SOSP '71, pages 95–100, Palo Alto, California, USA, 1971. Association for Computing Machinery.

[Edg13] Jake Edge. The future of realtime Linux, November 2013. URL: `https://lwn.net/Articles/572740/`.

[Edg14] Jake Edge. The future of the realtime patch set, October 2014. URL: `https://lwn.net/Articles/617140/`.

[EGCD03] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC language specifications v1.1, May 2003. URL: `http://upc.gwu.edu` [broken, February 27, 2021].

[EGMdB11] Stephane Eranian, Eric Gouriou, Tipp Moseley, and Willem de Bruijn. Linux kernel profiling with perf, June 2011. `https://perf.wiki.kernel.org/index.php/Tutorial`.

[Ell80] Carla Schlatter Ellis. Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers*, C-29(9):811–817, September 1980.

[ELLM07] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. SNZI: scalable NonZero indicators. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 13–22, Portland, Oregon, USA, 2007. ACM.

[EMV+20a] Marco Elver, Paul E. McKenney, Dmitry Vyukov, Andrey Konovalov, Alexander Potapenko, Kostya Serebryany, Alan Stern, Andrea Parri, Akira Yokosawa, Peter Zijlstra, Will Deacon, Daniel Lustig, Boqun Feng, Joel Fernandes, Jade Alglave, and Luc Maranget. Concurrency bugs should fear the big bad data-race detector (part 1), April 2020. Linux Weekly News.

[EMV+20b] Marco Elver, Paul E. McKenney, Dmitry Vyukov, Andrey Konovalov, Alexander Potapenko, Kostya Serebryany, Alan Stern, Andrea Parri, Akira Yokosawa, Peter Zijlstra, Will Deacon, Daniel Lustig, Boqun Feng, Joel Fernandes, Jade Alglave, and Luc Maranget. Concurrency bugs should fear the big bad data-race detector (part 2), April 2020. Linux Weekly News.

[Eng68] Douglas Engelbart. The demo, December 1968. URL: `http://thedemo.org/`.

www.dbooks.org

[ENS05] Ryan Eccles, Blair Nonneck, and Deborah A. Stacey. Exploring parallel programming knowledge in the novice. In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 97–102, Guelph, Ontario, Canada, 2005. IEEE Computer Society.

[Eri08] Christer Ericson. Aiding pathfinding with cellular automata, June 2008. `http://realtimecollisiondetection.net/blog/?p=57`.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[ES05] Ryan Eccles and Deborah A. Stacey. Understanding the parallel programmer. In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 156–160, Guelph, Ontario, Canada, 2005. IEEE Computer Society.

[ETH11] ETH Zurich. Parallel solver for a perfect maze, March 2011. URL: `http://nativesystems.inf.ethz.ch/pub/Main/WebHomeLecturesParallelProgrammingExercises/pp2011hw04.pdf` [broken, November 2016].

[Eva11] Jason Evans. Scalable memory allocation using jemalloc, January 2011. `https://engineering.fb.com/2011/01/03/core-data/scalable-memory-allocation-using-jemalloc/`.

[Fel50] W. Feller. *An Introduction to Probability Theory and its Applications*. John Wiley, 1950.

[Fen73] J. Fennel. Instruction selection in a two-program counter instruction unit. Technical Report US Patent 3,728,692, Assigned to International Business Machines Corp, Washington, DC, April 1973.

[Fen15] Boqun Feng. powerpc: Make value-returning atomics fully ordered, November 2015. Git commit: `https://git.kernel.org/linus/49e9cf3f0c04`.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):1–61, 2007.

[FIMR16] Pascal Felber, Shady Issa, Alexander Matveev, and Paolo Romano. Hardware read-write lock elision. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, London, United Kingdom, 2016. Association for Computing Machinery.

[Fos10] Ron Fosner. Scalable multithreaded programming with tasks. *MSDN Magazine*, 2010(11):60–69, November 2010. `http://msdn.microsoft.com/en-us/magazine/gg309176.aspx`.

[FPB79] Jr. Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1979.

[Fra04] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[FRK02] Hubertus Francke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium*, pages 479–495, June 2002. Available: `https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf` [Viewed May 22, 2011].

[FSP+17] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell.

Mixed-size concurrency: ARM, POWER, C/C++11, and SC. *SIGPLAN Not.*, 52(1):429–442, January 2017.

[GAJM15] Alex Groce, Iftekhar Ahmed, Carlos Jensen, and Paul E. McKenney. How verified is my code? falsification-driven verification (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 737–748, Washington, DC, USA, 2015. IEEE Computer Society.

[Gar90] Arun Garg. Parallel STREAMS: a multi-processor implementation. In *USENIX Conference Proceedings*, pages 163–176, Berkeley CA, February 1990. USENIX Association. Available: `https://archive.org/details/1990-proceedings-winter-dc/page/163/mode/2up`.

[Gar07] Bryan Gardiner. IDF: Gordon Moore predicts end of Moore's law (again), September 2007. Available: `https://www.wired.com/2007/09/idf-gordon-mo-1/` [Viewed: February 27, 2021].

[GC96] Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 123–136, Seattle, WA, October 1996. USENIX Association.

[GDZE10] Olga Golovanevsky, Alon Dayan, Ayal Zaks, and David Edelsohn. Trace-based data layout optimizations for multi-core processors. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 81–95, Pisa, Italy, 2010. Springer-Verlag.

[GG14] Vincent Gramoli and Rachid Guerraoui. Democratizing transactional programming. *Commun. ACM*, 57(1):86–93, January 2014.

[GGK18] Christina Giannoula, Georgios Goumas, and Nectarios Koziris. Combining HTM with RCU to speed up graph coloring on multicore platforms. In Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis, editors, *High Performance Computing*, pages 350–369, Cham, 2018. Springer International Publishing.

[GGL+19] Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis. Lock–unlock: Is that all? a pragmatic analysis of locking in software systems. *ACM Trans. Comput. Syst.*, 36(1):1:1–1:149, March 2019.

[Gha95] Kourosh Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical Report CSL-TR-95-685, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA, December 1995. Available: `https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-9.pdf` [Viewed: October 11, 2004].

[GHH+14] Alex Groce, Klaus Havelund, Gerard J. Holzmann, Rajeev Joshi, and Ru-Gang Xu. Establishing flight software reliability: testing, model checking, constraint-solving, monitoring and learning. *Ann. Math. Artif. Intell.*, 70(4):315–349, 2014.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

www.dbooks.org

[GKAS99] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, pages 87–100, New Orleans, LA, February 1999.

[GKP13] Justin Gottschlich, Rob Knauerhase, and Gilles Pokam. But how do we really debug transactional memory? In *5th USENIX Workshop on Hot Topics in Parallelism (HotPar 2013)*, San Jose, CA, USA, June 2013.

[GKPS95] Ben Gamsa, Orran Krieger, E. Parsons, and Michael Stumm. Performance issues for multiprocessor operating systems, November 1995. Technical Report CSRI-339, Available: `ftp://ftp.cs.toronto.edu/pub/reports/csri/339/339.ps`.

[Gla18] Stjepan Glavina. Merge remaining subcrates, November 2018. `https://github.com/crossbeam-rs/crossbeam/commit/d9b1e3429450a64b490f68c08bd191417e68f00c`.

[Gle10] Thomas Gleixner. Realtime linux: academia v. reality, July 2010. URL: `https://lwn.net/Articles/397422/`.

[Gle12] Thomas Gleixner. Linux -rt kvm guest demo, December 2012. Personal communication.

[GMTW08] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, May 2008.

[Gol18a] David Goldblatt. Add the Seq module, a simple seqlock implementation, April 2018. `https://github.com/jemalloc/jemalloc/tree/06a8c40b36403e902748d3f2a14e6dd43488ae89`.

[Gol18b] David Goldblatt. P1202: Asymmetric fences, October 2018. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1202r0.pdf`.

[Gol19] David Goldblatt. There might not be an elegant OOTA fix, October 2019. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1916r0.pdf`.

[GPB+07] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java: Concurrency in Practice*. Addison Wesley, Upper Saddle River, NJ, USA, 2007.

[Gra91] Jim Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, 1991.

[Gra02] Jim Gray. Super-servers: Commodity computer clusters pose a software challenge, April 2002. Available: `http://research.microsoft.com/en-us/um/people/gray/papers/superservers(4t_computers).doc` [Viewed: June 23, 2004].

[Gre19] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley Professional, 1st edition, 2019.

[Gri00] Scott Griffen. Internet pioneers: Doug englebart, May 2000. Available: `https://www.ibiblio.org/pioneers/englebart.html` [Viewed November 28, 2008].

[Gro01] The Open Group. Single UNIX specification, July 2001. `http://www.opengroup.org/onlinepubs/007908799/index.html`.

[Gro07] Dan Grossman. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 695–706, Montreal, Quebec, Canada, October 2007. ACM. Available: `https://homes.cs.washington.edu/~djg/papers/analogy_oopsla07.pdf` [Viewed February 27, 2021].

[GRY12] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. Verifying highly concurrent algorithms with grace (extended version), July 2012. `https://software.imdea.org/~gotsman/papers/recycling-esop13-ext.pdf`.

[GRY13] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP'13: European Symposium on Programming*, pages 249–269, Rome, Italy, 2013. Springer-Verlag.

[GT90] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[Gui18] Hugo Guiroux. *Understanding the performance of mutual exclusion algorithms on modern multicore machines*. PhD thesis, Université Grenoble Alpes, 2018. `https://hugoguiroux.github.io/assets/these.pdf`.

[Gwy15] David Gwynne. introduce srp, which according to the manpage i wrote is short for "shared reference pointers"., July 2015. `https://github.com/openbsd/src/blob/HEAD/sys/kern/kern_srp.c`.

[GYW+19] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 913–928, Renton, WA, USA, 2019. USENIX Association.

[Har16] "No Bugs" Hare. Infographics: Operation costs in CPU clock cycles, September 2016. `http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/`.

[Hay20] Timothy Hayes. A shift to concurrency, October 2020. `https://community.arm.com/developer/research/b/articles/posts/arms-transactional-memory-extension-support-`.

[HCS+05] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, and Victor Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, Seattle, WA, USA, 2005. IEEE Computer Society.

[Hei27] W. Heisenberg. Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik. *Zeitschrift für Physik*, 43(3-4):172–198, 1927. English translation in "Quantum theory and measurement" by Wheeler and Zurek.

[Her90] Maurice P. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, Seattle, WA, USA, March 1990.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, January 1991.

www.dbooks.org

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[Her05] Maurice Herlihy. The transactional manifesto: software engineering and non-blocking synchronization. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 280–280, Chicago, IL, USA, 2005. ACM Press.

[Her11] Benjamin Herrenschmidt. powerpc: Fix atomic_xxx_return barrier semantics, November 2011. Git commit: `https://git.kernel.org/linus/b97021f85517`.

[HHK+13] A. Haas, T.A. Henzinger, C.M. Kirsch, M. Lippautz, H. Payer, A. Sezgin, and A. Sokolova. Distributed queues in shared memory—multicore performance and scalability through quantitative relaxation. In *Proc. International Conference on Computing Frontiers*, Ischia, Italy, 2013. ACM.

[HKLP12] Andreas Haas, Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. How FIFO is your concurrent FIFO queue? In *Proceedings of the Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, Tucson, AZ USA, October 2012.

[HL86] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research*. Holden-Day, 1986.

[HLM02] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing*, pages 339–353, Toulouse, France, October 2002.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 73–82, Providence, RI, May 2003. The Institute of Electrical and Electronics Engineers, Inc.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proceeding of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, USA, May 1993.

[HMB06] Thomas E. Hart, Paul E. McKenney, and Angela Demke Brown. Making lockless synchronization fast: Performance implications of memory reclamation. In *20th IEEE International Parallel and Distributed Processing Symposium*, Rhodes, Greece, April 2006. Available: `http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf` [Viewed April 28, 2008].

[HMBW07] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007.

[HMDZ06] David Howells, Paul E. McKenney, Will Deacon, and Peter Zijlstra. Linux kernel memory barriers, March 2006. `https://www.kernel.org/doc/Documentation/memory-barriers.txt`.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hol03] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, MA, USA, 2003.

[Hor18] Jann Horn. Reading privileged memory with a side-channel, January 2018. `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`.

[HOS89] James P. Hennessy, Damian L. Osisek, and Joseph W. Seigh II. Passive serialization in a multitasking environment. Technical Report US Patent 4,809,168, Assigned to International Business Machines Corp, Washington, DC, February 1989.

[How12] Phil Howard. *Extending Relativistic Programming to Multiple Writers*. PhD thesis, Portland State University, 2012.

[HP95] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 1995.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Fifth Edition*. Morgan Kaufman, 2011.

[HP17] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Sixth Edition*. Morgan Kaufman, 2017.

[Hra13] Adam Hraška. Read-copy-update for helenos. Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Distributed and Dependable Systems, 2013.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Burlington, MA, USA, 2008.

[HSLS20] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming, 2nd Edition*. Morgan Kaufmann, Burlington, MA, USA, 2020.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[HW92] Wilson C. Hsieh and William E. Weihl. Scalable reader-writer locks for parallel systems. In *Proceedings of the 6th International Parallel Processing Symposium*, pages 216–230, Beverly Hills, CA, USA, March 1992.

[HW11] Philip W. Howard and Jonathan Walpole. A relativistic enhancement to software transactional memory. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism*, HotPar'11, pages 1–6, Berkeley, CA, 2011. USENIX Association.

[HW13] Philip W. Howard and Jonathan Walpole. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2013.

[IBM94] IBM Microelectronics and Motorola. *PowerPC Microprocessor Family: The Programming Environments*, 1994.

[Inm85] Jack Inman. Implementing loosely coupled functions on tightly coupled engines. In *USENIX Conference Proceedings*, pages 277–298, Portland, OR, June 1985. USENIX Association.

[Inm07] Bill Inmon. Time value of information, January 2007. URL: `http://www.b-eye-network.com/view/3365` [broken, February 2021].

www.dbooks.org

[Int92] International Standards Organization. *Information Technology - Database Language SQL*. ISO, 1992. Available (Second informal review draft of ISO/IEC 9075:1992): `http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt` [Viewed September 19, 2008].

[Int02a] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 3: Instruction Set Reference*, 2002.

[Int02b] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 3: System Architecture*, 2002.

[Int04a] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, 2004.

[Int04b] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2004.

[Int04c] International Business Machines Corporation. z/Architecture principles of operation, May 2004. Available: `http://publibz.boulder.ibm.com/epubs/pdf/dz9zr003.pdf` [Viewed: February 16, 2005].

[Int07] Intel Corporation. *Intel 64 Architecture Memory Ordering White Paper*, 2007.

[Int11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, 2011. Available: `http://www.intel.com/Assets/PDF/manual/253668.pdf` [Viewed: February 12, 2011].

[Int16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, 2016.

[Int20] Intel Corporation. *Intel Transactional Synchronization Extensions (Intel TSX) Programming Considerations*, 2021.1 edition, December 2020. In *Intel C++ Compiler Classic Developer Guide and Reference*, `https://software.intel.com/content/dam/develop/external/us/en/documents/cpp_compiler_classic.pdf`, page 1506.

[Jac88] Van Jacobson. Congestion avoidance and control. In *SIGCOMM '88*, pages 314–329, August 1988.

[Jac93] Van Jacobson. Avoid read-side locking via delayed free, September 1993. private communication.

[Jac08] Daniel Jackson. MapReduce course, January 2008. Available: `https://sites.google.com/site/mriap2008/` [Viewed January 3, 2013].

[Jef14] Alan Jeffrey. Jmm revision status, July 2014. `https://mail.openjdk.java.net/pipermail/jmm-dev/2014-July/000072.html`.

[JLK16a] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization (KASLR) with Intel TSX, July 2016. Black Hat USA 2018 `https://www.blackhat.com/us-16/briefings.html#breaking-kernel-address-space-layout-randomization-kaslr-with-intel-tsx`.

[JLK16b] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with Intel TSX. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, Vienna, Austria, 2016. ACM.

[JMRR02] Benedict Joseph Jackson, Paul E. McKenney, Ramakrishnan Rajamony, and Ronald Lynn Rockhold. Scalable interruptible queue locks for shared-memory multiprocessor. US Patent 6,473,819, Assigned to International Business Machines Corporation, Washington, DC, October 2002.

[Joh77] Stephen Johnson. Lint, a C program checker, December 1977. Computer Science Technical Report 65, Bell Laboratories.

[Joh95] Aju John. Dynamic vnodes – design and implementation. In *USENIX Winter 1995*, pages 11–23, New Orleans, LA, January 1995. USENIX Association. Available: `https://www.usenix.org/publications/library/proceedings/neworl/full_papers/john.a` [Viewed October 1, 2010].

[Jon11] Dave Jones. Trinity: A system call fuzzer. In *Proceedings of the 13th Ottawa Linux Symposium*, pages ???–???, Ottawa, Canada, June 2011. Project repository: `https://github.com/kernelslacker/trinity`.

[JSG12] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for IBM System z, December 2012. The 45th Annual IEEE/ACM International Symposium on MicroArchitecture, URL: `https://www.microarch.org/micro45/talks-posters/3-jacobi-presentation.pdf`.

[Kaa15] Frans Kaashoek. Parallel computing and the os. In *SOSP History Day*, October 2015.

[KCH+06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kumar, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming*, New York, New York, United States, 2006. ACM SIGPLAN.

[KDI20] Alex Kogan, Dave Dice, and Shady Issa. Scalable range locks for scalable address spaces and beyond. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, Heraklion, Greece, 2020. Association for Computing Machinery.

[Kel17] Michael J. Kelly. How might the manufacturability of the hardware at device level impact on exascale computing?, 2017. Keynote speech at Multicore World 2017, URL: `https://openparallel.com/multicore-world-2017/program-2017/abstracts2017/`.

[Ken20] Chris Kennelly. TCMalloc overview, February 2020. `https://google.github.io/tcmalloc/overview.html`.

[KFC11] KFC. Memristor processor solves mazes, March 2011. URL: `https://www.technologyreview.com/2011/03/03/196572/memristor-processor-solves-mazes/`.

[Khi14] Maxim Khizhinsky. Memory management schemes, June 2014. `https://kukuruku.co/post/lock-free-data-structures-the-inside-memory-management-schemes/`.

[Khi15] Max Khiszinsky. Lock-free data structures. the inside. RCU, February 2015. `https://kukuruku.co/post/lock-free-data-structures-the-inside-rcu/`.

[Kis14] Jan Kiszka. Real-time virtualization - how crazy are we? In *Linux Plumbers Conference*, Duesseldorf, Germany, October 2014. URL: `https://blog.linuxplumbersconf.org/2014/ocw/proposals/1935`.

www.dbooks.org

[Kiv13] Avi Kivity. rcu: add basic read-copy-update implementation, August 2013. https://github.com/cloudius-systems/osv/commit/94b69794fb9e6c99d78ca9a58ddaee1c31256b43.

[Kiv14a] Avi Kivity. rcu hashtable, July 2014. https://github.com/cloudius-systems/osv/commit/7fa2728e5d03b2174b4a39d94b21940d11926e90.

[Kiv14b] Avi Kivity. rcu: introduce an rcu list type, April 2014. https://github.com/cloudius-systems/osv/commit/4e46586093aeaf339fef8e08d123a6f6b0abde5b.

[KL80] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, September 1980.

[Kle14] Andi Kleen. Scaling existing lock-based applications with lock elision. *Commun. ACM*, 57(3):52–56, March 2014.

[Kle17] Matt Klein. Envoy threading model, July 2017. https://blog.envoyproxy.io/envoy-threading-model-a8d44b922310.

[KLP12] Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. Fast and scalable k-FIFO queues. Technical Report 2012-04, University of Salzburg, Salzburg, Austria, June 2012.

[KM13] Konstantin Khlebnikov and Paul E. McKenney. RCU: non-atomic assignment to long/pointer variables in gcc, January 2013. https://lore.kernel.org/lkml/50F52FC8.4000701@openvz.org/.

[KMK+19] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. Mv-rlu: Scaling read-log-update with multi-versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 779–792, Providence, RI, USA, 2019. ACM.

[Kni86] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 105–112, Cambridge, Massachusetts, USA, 1986. ACM.

[Kni08] John U. Knickerbocker. 3D chip technology. *IBM Journal of Research and Development*, 52(6), November 2008. URL: http://www.research.ibm.com/journal/rd52-6.html [Link to each article is broken as of November 2016; Available via https://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=5388557].

[Knu73] Donald Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.

[Kra17] Vlad Krasnov. On the dangers of Intel's frequency scaling, November 2017. https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[KS17a] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the linux kernel's hierarchical read-copy update (Tree RCU). Technical report, National Technical University of Athens, January 2017. https://github.com/michalis-/rcu/blob/master/rcupaper.pdf.

[KS17b] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the Linux kernel's hierarchical read-copy-update (Tree RCU). In *Proceedings of International SPIN Symposium on Model Checking of Software*, SPIN 2017, New York, NY, USA, July 2017. ACM.

[KS19] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the linux kernel's read—copy update (RCU). *Int. J. Softw. Tools Technol. Transf.*, 21(3):287âĂŞ306, June 2019.

[KWS97] Leonidas Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.

[LA94] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS VI, pages 25–35, San Jose, California, USA, October 1994. ACM.

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[Lam77] Leslie Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, November 1977.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[LBD+04] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Softw.*, 21(3):92âĂŞ100, May 2004.

[Lea97] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley Longman, Reading, MA, USA, 1997.

[Lem18] Daniel Lemire. AVX-512: when and how to use these new instructions, September 2018. `https://lemire.me/blog/2018/09/07/avx-512-when-and-how-to-use-these-new-instructions/`.

[LGW+15] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the ibm power8 processor. *IBM J. Res. Dev.*, 59(1):8:1–8:14, January 2015.

[LHF05] Michael Lyons, Bill Hay, and Brad Frey. PowerPC storage model and AIX programming, November 2005. `http://www.ibm.com/developerworks/systems/articles/powerpc.html`.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Commun. ACM*, 31(3):300–312, 1988.

[LLO09] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 101–110, Calgary, AB, Canada, 2009. ACM.

[LLS13] Yujie Liu, Victor Luchangco, and Michael Spear. Mindicators: A scalable approach to quiescence. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 206–215, Washington, DC, USA, 2013. IEEE Computer Society.

www.dbooks.org

[LMKM16] Lihao Liang, Paul E. McKenney, Daniel Kroening, and Tom Melham. Verification of the tree-based hierarchical read-copy update in the Linux kernel. Technical report, Cornell University Library, October 2016. https://arxiv.org/abs/1610.03052.

[LMKM18] Lihao Liang, Paul E. McKenney, Daniel Kroening, and Tom Melham. Verification of tree-based hierarchical Read-Copy Update in the Linux Kernel. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19–23, 2018*, 2018.

[Loc02] Doug Locke. Priority inheritance: The real story, July 2002. URL: http://www.linuxdevices.com/articles/AT5698775833.html [broken, November 2016], page capture available at https://www.math.unipd.it/%7Etullio/SCD/2007/Materiale/Locke.pdf.

[Lom77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGSOFT Softw. Eng. Notes*, 2(2):128–137, 1977. URL: http://portal.acm.org/citation.cfm?id=808319#.

[LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, 1980.

[LS86] Vladimir Lanin and Dennis Shasha. A symmetric concurrent b-tree algorithm. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 380–389, Dallas, Texas, United States, 1986. IEEE Computer Society Press.

[LS11] Yujie Liu and Michael Spear. Toxic transactions. In *TRANSACT 2011*, San Jose, CA, USA, June 2011. ACM SIGPLAN.

[LSLK14] Carl Leonardsson, Kostis Sagonas, Truc Nguyen Lam, and Michalis Kokologiannakis. Nidhugg, July 2014. https://github.com/nidhugg/nidhugg.

[LVK+17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. *SIGPLAN Not.*, 52(6):618–632, June 2017.

[LZC14] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 219–230, Philadelphia, PA, June 2014. USENIX Association.

[MAK+01] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001. URL: https://www.kernel.org/doc/ols/2001/read-copy.pdf, http://www.rdrop.com/users/paulmck/RCU/rclock_OLS.2001.05.01c.pdf.

[Mar17] Luc Maraget. Aarch64 model vs. hardware, May 2017. http://pauillac.inria.fr/~maranget/cats7/model-aarch64/specific.html.

[Mar18] Catalin Marinas. Queued spinlocks model, March 2018. https://git.kernel.org/pub/scm/linux/kernel/git/cmarinas/kernel-tla.git.

[Mas92] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, New York, NY, 1992.

[Mat17] Norm Matloff. *Programming on Parallel Machines*. University of California, Davis, Davis, CA, USA, 2017.

[MB20] Paul E. McKenney and Hans Boehm. P2055R0: A relaxed guide to memory_order_relaxed, January 2020. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2055r0.pdf.

[MBM+06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture (HPCA-12)*, Austin, Texas, United States, 2006. IEEE. Available: http://www.cs.wisc.edu/multifacet/papers/hpca06_logtm.pdf [Viewed December 21, 2006].

[MBWW12] Paul E. McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. RCU usage in the linux kernel: One decade later, September 2012. Technical report paulmck.2012.09.17, http://rdrop.com/users/paulmck/techreports/survey.2012.09.17a.pdf.

[McK90] Paul E. McKenney. Stochastic fairness queuing. In *IEEE INFOCOM'90 Proceedings*, pages 733–740, San Francisco, June 1990. The Institute of Electrical and Electronics Engineers, Inc. Revision available: http://www.rdrop.com/users/paulmck/scalability/paper/sfq.2002.06.04.pdf [Viewed May 26, 2008].

[McK95] Paul E. McKenney. Differential profiling. In *MASCOTS 1995*, pages 237–241, Toronto, Canada, January 1995.

[McK96a] Paul E. McKenney. *Pattern Languages of Program Design*, volume 2, chapter 31: Selecting Locking Designs for Parallel Programs, pages 501–531. Addison-Wesley, June 1996. Available: http://www.rdrop.com/users/paulmck/scalability/paper/mutexdesignpat.pdf [Viewed February 17, 2005].

[McK96b] Paul E. McKenney. Selecting locking primitives for parallel programs. *Communications of the ACM*, 39(10):75–82, October 1996.

[McK99] Paul E. McKenney. Differential profiling. *Software - Practice and Experience*, 29(3):219–234, 1999.

[McK01] Paul E. McKenney. RFC: patch to allow lock-free traversal of lists with insertion, October 2001. Available: https://lore.kernel.org/lkml/200110090155.f991tPt22329@eng4.beaverton.ibm.com/ [Viewed January 05, 2021].

[McK03] Paul E. McKenney. Using RCU in the Linux 2.5 kernel. *Linux Journal*, 1(114):18–26, October 2003. Available: https://www.linuxjournal.com/article/6993 [Viewed November 14, 2007].

[McK04] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[McK05a] Paul E. McKenney. Memory ordering in modern microprocessors, part I. *Linux Journal*, 1(136):52–57, August 2005. Available: https://www.linuxjournal.com/article/8211 http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.19a.pdf [Viewed November 30, 2007].

[McK05b] Paul E. McKenney. Memory ordering in modern microprocessors, part II. *Linux Journal*, 1(137):78–82, September 2005. Available: https://www.

www.dbooks.org

linuxjournal.com/article/8212 http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.19a.pdf [Viewed November 30, 2007].

[McK05c] Paul E. McKenney. A realtime preemption overview, August 2005. URL: https://lwn.net/Articles/146861/.

[McK06] Paul E. McKenney. Sleepable RCU, October 2006. Available: https://lwn.net/Articles/202847/ Revised: http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf [Viewed August 21, 2006].

[McK07a] Paul E. McKenney. The design of preemptible read-copy-update, October 2007. Available: https://lwn.net/Articles/253651/ [Viewed October 25, 2007].

[McK07b] Paul E. McKenney. Immunize rcu_dereference() against crazy compiler writers, October 2007. Git commit: https://git.kernel.org/linus/97b430320ce7.

[McK07c] Paul E. McKenney. [PATCH] QRCU with lockless fastpath, February 2007. Available: https://lkml.org/lkml/2007/2/25/18 [Viewed March 27, 2008].

[McK07d] Paul E. McKenney. Priority-boosting RCU read-side critical sections, February 2007. https://lwn.net/Articles/220677/.

[McK07e] Paul E. McKenney. RCU and unloadable modules, January 2007. Available: https://lwn.net/Articles/217484/ [Viewed November 22, 2007].

[McK07f] Paul E. McKenney. Using Promela and Spin to verify parallel algorithms, August 2007. Available: https://lwn.net/Articles/243851/ [Viewed September 8, 2007].

[McK08a] Paul E. McKenney. Hierarchical RCU, November 2008. https://lwn.net/Articles/305782/.

[McK08b] Paul E. McKenney. rcu: fix rcu_try_flip_waitack_needed() to prevent grace-period stall, May 2008. Git commit: https://git.kernel.org/linus/d7c0651390b6.

[McK08c] Paul E. McKenney. rcu: fix misplaced mb() in rcu_enter/exit_nohz(), March 2008. Git commit: https://git.kernel.org/linus/ae66be9b71b1.

[McK08d] Paul E. McKenney. RCU part 3: the RCU API, January 2008. Available: https://lwn.net/Articles/264090/ [Viewed January 10, 2008].

[McK08e] Paul E. McKenney. "Tree RCU": scalable classic RCU implementation, December 2008. Git commit: https://git.kernel.org/linus/64db4cfff99c.

[McK08f] Paul E. McKenney. What is RCU? part 2: Usage, January 2008. Available: https://lwn.net/Articles/263130/ [Viewed January 4, 2008].

[McK09a] Paul E. McKenney. Re: [PATCH fyi] RCU: the bloatwatch edition, January 2009. Available: https://lkml.org/lkml/2009/1/14/449 [Viewed January 15, 2009].

[McK09b] Paul E. McKenney. Transactional memory everywhere?, September 2009. https://paulmck.livejournal.com/13841.html.

[McK11a] Paul E. McKenney. 3.0 and RCU: what went wrong, July 2011. https://lwn.net/Articles/453002/.

[McK11b] Paul E. McKenney. Concurrent code and expensive instructions, January 2011. Available: https://lwn.net/Articles/423994 [Viewed January 28, 2011].

[McK11c] Paul E. McKenney. Transactional memory everywhere: Htm and cache geometry, June 2011. https://paulmck.livejournal.com/tag/transactional%20memory%20everywhere.

[McK11d] Paul E. McKenney. Validating memory barriers and atomic instructions, December 2011. https://lwn.net/Articles/470681/.

[McK11e] Paul E. McKenney. Verifying parallel software: Can theory meet practice?, January 2011. http://www.rdrop.com/users/paulmck/scalability/paper/VericoTheoryPractice.2011.01.28a.pdf.

[McK12a] Paul E. McKenney. Beyond expert-only parallel programming? In *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, RACES '12, pages 25–32, Tucson, Arizona, USA, 2012. ACM.

[McK12b] Paul E. McKenney. Making RCU safe for battery-powered devices, February 2012. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdynticks.2012.02.15b.pdf [Viewed March 1, 2012].

[McK12c] Paul E. McKenney. Retrofitted parallelism considered grossly sub-optimal. In *4th USENIX Workshop on Hot Topics on Parallelism*, page 7, Berkeley, CA, USA, June 2012.

[McK12d] Paul E. McKenney. Signed overflow optimization hazards in the kernel, August 2012. https://lwn.net/Articles/511259/.

[McK12e] Paul E. McKenney. Transactional memory everywhere: Hardware transactional lock elision, May 2012. Available: https://paulmck.livejournal.com/32267.html [Viewed January 28, 2021].

[McK13] Paul E. McKenney. Structured deferral: synchronization via procrastination. *Commun. ACM*, 56(7):40–49, July 2013.

[McK14a] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It? (First Edition)*. kernel.org, Corvallis, OR, USA, 2014.

[McK14b] Paul E. McKenney. N4037: Non-transactional implementation of atomic tree move, May 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4037.pdf.

[McK14c] Paul E. McKenney. Proper care and feeding of return values from rcu_dereference(), February 2014. https://www.kernel.org/doc/Documentation/RCU/rcu_dereference.txt.

[McK14d] Paul E. McKenney. The RCU API, 2014 edition, September 2014. https://lwn.net/Articles/609904/.

[McK14e] Paul E. McKenney. Recent read-mostly research, November 2014. https://lwn.net/Articles/619355/.

[McK15a] Paul E. McKenney. Formal verification and Linux-kernel concurrency. In *Compositional Verification Methods for Next-Generation Concurrency*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2015. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

www.dbooks.org

[McK15b] Paul E. McKenney. [PATCH tip/core/rcu 01/10] rcu: Make rcu_nmi_enter() handle nesting, January 2015. https://lore.kernel.org/lkml/1420651257-553-1-git-send-email-paulmck@linux.vnet.ibm.com/.

[McK15c] Paul E. McKenney. Practical experience with formal verification tools. In *Verified Trustworthy Software Systems Specialist Meeting*. The Royal Society, April 2015. http://www.rdrop.com/users/paulmck/scalability/paper/Validation.2016.04.06e.SpecMtg.pdf.

[McK15d] Paul E. McKenney. RCU requirements part 2 — parallelism and software engineering, August 2015. https://lwn.net/Articles/652677/.

[McK15e] Paul E. McKenney. RCU requirements part 3, August 2015. https://lwn.net/Articles/653326/.

[McK15f] Paul E. McKenney. Re: [patch tip/locking/core v4 1/6] powerpc: atomic: Make *xchg and *cmpxchg a full barrier, October 2015. Email thread: https://lore.kernel.org/lkml/20151014201916.GB3910@linux.vnet.ibm.com/.

[McK15g] Paul E. McKenney. Requirements for RCU part 1: the fundamentals, July 2015. https://lwn.net/Articles/652156/.

[McK16] Paul E. McKenney. Beyond the Issaquah challenge: High-performance scalable complex updates, September 2016. http://www2.rdrop.com/users/paulmck/RCU/Updates.2016.09.19i.CPPCON.pdf.

[McK17] Paul E. McKenney. Verification challenge 6: Linux-kernel Tree RCU, June 2017. https://paulmck.livejournal.com/46993.html.

[McK19a] Paul E. McKenney. A critical RCU safety property is... Ease of use! In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 132–143, Haifa, Israel, 2019. ACM.

[McK19b] Paul E. McKenney. The RCU API, 2019 edition, January 2019. https://lwn.net/Articles/777036/.

[McK19c] Paul E. McKenney. RCU's first-ever CVE, and how i lived to tell the tale, January 2019. linux.conf.au Slides: http://www.rdrop.com/users/paulmck/RCU/cve.2019.01.23e.pdf Video: https://www.youtube.com/watch?v=hZX1aokdNiY.

[MCM02] Paul E. McKenney, Kevin A. Closson, and Raghupathi Malige. Lingering locks with fairness control for multi-node computer systems. US Patent 6,480,918, Assigned to International Business Machines Corporation, Washington, DC, November 2002.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *Transactions of Computer Systems*, 9(1):21–65, February 1991.

[MD92] Paul E. McKenney and Ken F. Dove. Efficient demultiplexing of incoming tcp packets. In *SIGCOMM '92, Proceedings of the Conference on Communications Architecture & Protocols*, pages 269–279, Baltimore, MD, August 1992. Association for Computing Machinery.

[MDJ13a] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. URCU-protected hash tables, November 2013. https://lwn.net/Articles/573431/.

[MDJ13b] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. URCU-protected queues and stacks, November 2013. https://lwn.net/Articles/573433/.

[MDJ13c] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. User-space RCU, November 2013. https://lwn.net/Articles/573424/.

[MDR16] Paul E. McKenney, Will Deacon, and Luis R. Rodriguez. Semantics of MMIO mapping attributes across architectures, August 2016. https://lwn.net/Articles/698014/.

[MDSS20] Hans Meuer, Jack Dongarra, Erich Strohmaier, and Horst Simon. Top 500: The list, November 2020. Available: https://top500.org/lists/ [Viewed March 6, 2021].

[Men16] Alexis Menard. Move OneWriterSeqLock and SharedMemorySeqLockBuffer from content/ to device/base/synchronization, September 2016. https://source.chromium.org/chromium/chromium/src/+/b39a3082846d5877a15e8b7e18d66cb142abe8af.

[Mer11] Rick Merritt. IBM plants transactional memory in CPU, August 2011. EE Times https://www.eetimes.com/ibm-plants-transactional-memory-in-cpu/.

[Met99] Panagiotis Takis Metaxas. Fast dithering on a data-parallel computer. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 570–576, Cambridge, MA, USA, 1999. IASTED.

[MG92] Paul E. McKenney and Gary Graunke. Efficient buffer allocation on shared-memory multiprocessors. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, pages 194–199, Tucson, AZ, February 1992. The Institute of Electrical and Electronics Engineers, Inc.

[MGM+09] Paul E. McKenney, Manish Gupta, Maged M. Michael, Phil Howard, Joshua Triplett, and Jonathan Walpole. Is parallel programming hard, and if so, why? Technical Report TR-09-02, Portland State University, Portland, OR, USA, February 2009. URL: https://archives.pdx.edu/ds/psu/10386 [Viewed February 13, 2021].

[MHS12] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip coherence is here to stay. *Communications of the ACM*, 55(7):78–89, July 2012.

[Mic02] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, August 2002.

[Mic03] Maged M. Michael. Cas-based lock-free algorithm for shared deques. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 651–660. Springer, 2003.

[Mic04] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

www.dbooks.org

[Mic08] Microsoft. *FlushProcessWriteBuffers function*, 2008. https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-flushprocesswritebuffers.

[Mic18] Maged Michael. Rewrite from experimental, use of deterministic schedule, improvements, June 2018. Hazard pointers: https://github.com/facebook/folly/commit/d42832d2a529156275543c7fa7183e1321df605d.

[Mil06] David S. Miller. Re: [PATCH, RFC] RCU : OOM avoidance and lower latency, January 2006. Available: https://lkml.org/lkml/2006/1/7/22 [Viewed February 29, 2012].

[MJST16] Paul E. McKenney, Alan Jeffrey, Ali Sezgin, and Tony Tye. Out-of-thin-air execution is vacuous, July 2016. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0422r0.html.

[MK88] Marshall Kirk McKusick and Michael J. Karels. Design of a general purpose memory allocator for the 4.3BSD UNIX kernel. In *USENIX Conference Proceedings*, Berkeley CA, June 1988.

[MKM12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, Bern, Switzerland, 2012. ACM.

[ML82] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure. Technical Report 82-01-01, Department of Computer Science, University of Washington, Seattle, Washington, January 1982.

[ML84] Udi Manber and Richard E. Ladner. Concurrency control in a dynamic search structure. *ACM Transactions on Database Systems*, 9(3):439–455, September 1984.

[MLH94] Peter Magnusson, Anders Landin, and Erik Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical Report T94:07, Swedish Institute of Computer Science, Kista, Sweden, February 1994.

[MM00] Ingo Molnar and David S. Miller. brlock, March 2000. URL: http://kernel.nic.funet.fi/pub/linux/kernel/v2.3/patch-html/patch-2.3.49/linux_include_linux_brlock.h.html.

[MMM+20] Paul E. McKenney, Maged Michael, Jens Maurer, Peter Sewell, Martin Uecker, Hans Boehm, Hubert Tong, Niall Douglas, Thomas Rodgers, Will Deacon, Michael Wong, David Goldblatt, Kostya Serebryany, and Anthony Williams. P1726R4: Pointer lifetime-end zap, July 2020. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1726r4.pdf.

[MMS19] Paul E. McKenney, Maged Michael, and Peter Sewell. N2369: Pointer lifetime-end zap, April 2019. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2369.pdf.

[MMTW10] Paul E. McKenney, Maged M. Michael, Josh Triplett, and Jonathan Walpole. Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory. *ACM Operating Systems Review*, 44(3), July 2010.

[MMW07] Paul E. McKenney, Maged Michael, and Jonathan Walpole. Why the grass may not be greener on the other side: A comparison of locking vs. transactional memory. In *Programming Languages and Operating Systems*, pages 1–5, Stevenson, Washington, USA, October 2007. ACM SIGOPS.

[Mol05] Ingo Molnar. Index of /pub/linux/kernel/projects/rt, February 2005. URL: `https://www.kernel.org/pub/linux/kernel/projects/rt/`.

[Mol06] Ingo Molnar. Lightweight robust futexes, March 2006. Available: `https://www.kernel.org/doc/Documentation/robust-futexes.txt` [Viewed February 14, 2021].

[Moo03] Gordon Moore. No exponential is forever–but we can delay forever. In *IBM Academy of Technology 2003 Annual Meeting*, San Francisco, CA, October 2003.

[Mor07] Richard Morris. Sir Tony Hoare: Geek of the week, August 2007. `https://www.red-gate.com/simple-talk/opinion/geek-of-the-week/sir-tony-hoare-geek-of-the-week/`.

[MOZ09] Nicholas Mc Guire, Peter Odhiambo Okech, and Qingguo Zhou. Analysis of inherent randomness of the linux kernel. In *Eleventh Real Time Linux Workshop*, Dresden, Germany, September 2009.

[MP15a] Paul E. McKenney and Aravinda Prasad. Recent read-mostly research in 2015, December 2015. `https://lwn.net/Articles/667593/`.

[MP15b] Paul E. McKenney and Aravinda Prasad. Some more details on read-log-update, December 2015. `https://lwn.net/Articles/667720/`.

[MPA+06] Paul E. McKenney, Chris Purcell, Algae, Ben Schumin, Gaius Cornelius, Qwertyus, Neil Conway, Sbw, Blainster, Canis Rufus, Zoicon5, Anome, and Hal Eisen. Read-copy update, July 2006. `https://en.wikipedia.org/wiki/Read-copy-update`.

[MPI08] MPI Forum. Message passing interface forum, September 2008. Available: `http://www.mpi-forum.org/` [Viewed September 9, 2008].

[MR08] Paul E. McKenney and Steven Rostedt. Integrating and validating dynticks and preemptable RCU, April 2008. Available: `https://lwn.net/Articles/279077/` [Viewed April 24, 2008].

[MRP+17] Paul E. McKenney, Torvald Riegel, Jeff Preshing, Hans Boehm, Clark Nelson, Olivier Giroux, Lawrence Crowl, JF Bastian, and Michael Wong. Marking memory order consume dependency chains, February 2017. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0462r1.pdf`.

[MS93] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Conference Proceedings*, pages 295–306, Berkeley CA, February 1993. USENIX Association. Available: `http://www.rdrop.com/users/paulmck/scalability/paper/mpalloc.pdf` [Viewed January 30, 2005].

[MS95] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures, December 1995. Technical Report TR599.

[MS96] M.M Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc of the Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.

www.dbooks.org

[MS98a] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[MS98b] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998.

[MS01] Paul E. McKenney and Dipankar Sarma. Read-copy update mutual exclusion in Linux, February 2001. Available: `http://lse.sourceforge.net/locking/rcu/rcupdate_doc.html` [Viewed October 18, 2004].

[MS08] MySQL AB and Sun Microsystems. MySQL Downloads, November 2008. Available: `http://dev.mysql.com/downloads/` [Viewed November 26, 2008].

[MS09] Paul E. McKenney and Raul Silvera. Example POWER implementation for C/C++ memory model, February 2009. Available: `http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2009.02.27a.html` [Viewed: April 5, 2009].

[MS12] Alexander Matveev and Nir Shavit. Towards a fully pessimistic STM model. In *TRANSACT 2012*, San Jose, CA, USA, February 2012. ACM SIGPLAN.

[MS14] Paul E. McKenney and Alan Stern. Axiomatic validation of memory barriers and atomic instructions, August 2014. `https://lwn.net/Articles/608550/`.

[MS18] Luc Maranget and Alan Stern. lock.cat, May 2018. `https://github.com/torvalds/linux/blob/master/tools/memory-model/lock.cat`.

[MSA+02] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002. Available: `https://www.kernel.org/doc/ols/2002/ols2002-pages-338-367.pdf` [Viewed February 14, 2021].

[MSFM15] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 168–183, Monterey, California, 2015. ACM.

[MSK01] Paul E. McKenney, Jack Slingwine, and Phil Krueger. Experience with an efficient parallel kernel memory allocator. *Software – Practice and Experience*, 31(3):235–257, March 2001.

[MSM05] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison Wesley, Boston, MA, USA, 2005.

[MSS04] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling dcache with RCU. *Linux Journal*, 1(118):38–46, January 2004.

[MSS12] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models, October 2012. `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`.

[MT01] Jose F. Martinez and Josep Torrellas. Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. In *Workshop on Memory Performance Issues, International Symposium on Computer Architecture*, Gothenburg, Sweden, June 2001. Available: `https://iacoma.cs.uiuc.edu/iacoma-papers/wmpi_locks.pdf` [Viewed June 23, 2004].

[MT02] Jose F. Martinez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, San Jose, CA, October 2002.

[Mud01] Trevor Mudge. POWER: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, April 2001.

[Mus04] Museum Victoria Australia. CSIRAC: Australia's first computer, 2004. URL: `http://museumvictoria.com.au/csirac/`.

[MW05] Paul E. McKenney and Jonathan Walpole. RCU semantics: A first attempt, January 2005. Available: `http://www.rdrop.com/users/paulmck/RCU/rcu-semantics.2005.01.30a.pdf` [Viewed December 6, 2009].

[MW07] Paul E. McKenney and Jonathan Walpole. What is RCU, fundamentally?, December 2007. Available: `https://lwn.net/Articles/262464/` [Viewed December 27, 2007].

[MWB+17] Paul E. McKenney, Michael Wong, Hans Boehm, Jens Maurer, Jeffrey Yasskin, and JF Bastien. P0190R4: Proposal for new `memory_order_consume` definition, July 2017. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0190r4.pdf`.

[MWPF18] Paul E. McKenney, Ulrich Weigand, Andrea Parri, and Boqun Feng. Linux-kernel memory model, September 2018. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0124r6.html`.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. Wiley, 1979.

[NA18] Catherine E. Nemitz and James H. Anderson. Work-in-progress: Lock-based software transactional memory for real-time systems. In *2018 IEEE Real-Time Systems Symposium*, RTSS'18, pages 147–150, Nashville, TN, USA, 2018. IEEE.

[Nag18] Honnappa Nagarahalli. rcu: add RCU library supporting QSBR mechanism, May 2018. `https://git.dpdk.org/dpdk/tree/lib/librte_rcu`.

[Nes06a] Oleg Nesterov. Re: [patch] cpufreq: mark `cpufreq_tsc()` as `core_initcall_sync`, November 2006. Available: `https://lkml.org/lkml/2006/11/19/69` [Viewed May 28, 2007].

[Nes06b] Oleg Nesterov. Re: [rfc, patch 1/2] qrcu: "quick" srcu implementation, November 2006. Available: `https://lkml.org/lkml/2006/11/29/330` [Viewed November 26, 2008].

[NSHW20] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2020.

[NVi17a] NVidia. Accelerated computing — training, January 2017. `https://developer.nvidia.com/accelerated-computing-training`.

[NVi17b] NVidia. Existing university courses, January 2017. `https://developer.nvidia.com/educators/existing-courses`.

[O'H19] Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

www.dbooks.org

[OHOC20] Robert O'Callahan, Kyle Huey, Devon O'Dell, and Terry Coatta. To catch a failure: The record-and-replay approach to debugging: A discussion with robert oâĂŹcallahan, kyle huey, devon oâĂŹdell, and terry coatta. *Queue*, 18(1):61âĂŞ79, February 2020.

[ON07] Robert Olsson and Stefan Nilsson. TRASH: A dynamic LC-trie and hash data structure. In *Workshop on High Performance Switching and Routing (HPSR'07)*, May 2007.

[ONH+96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS VII*, Cambridge, MA, USA, October 1996.

[Ope97] Open Group. The single UNIX specification, version 2: Threads, 1997. Available: `http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html` [Viewed September 19, 2008].

[PAB+95] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 314–321, Copper Mountain, CO, December 1995.

[Pat10] David Patterson. The trouble with multicore. *IEEE Spectrum*, 2010:28–32, 52–53, July 2010.

[PAT11] V Pankratius and A R Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures (2011)*, SPAA '11, pages 43–52, San Jose, CA, USA, 2011. ACM.

[PBCE20] Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook. Evidence that computer science grades are not bimodal. *Commun. ACM*, 63(1):91–98, January 2020.

[PD11] Martin Pohlack and Stephan Diestelhorst. From lightweight hardware transactional memory to lightweight lock elision. In *TRANSACT 2011*, San Jose, CA, USA, June 2011. ACM SIGPLAN.

[Pen18] Roman Penyaev. [PATCH v2 01/26] introduce list_next_or_null_rr_rcu(), May 2018. `https://lkml.kernel.org/r/20180518130413.16997-2-roman.penyaev@profitbricks.com`.

[Pet06] Jeremy Peters. From reuters, automatic trading linked to news events, December 2006. URL: `http://www.nytimes.com/2006/12/11/technology/11reuters.html?ei=5088&en=e5e9416415a9eeb2&ex=1323493200...`

[Pig06] Nick Piggin. [patch 3/3] radix-tree: RCU lockless readside, June 2006. Available: `https://lkml.org/lkml/2006/6/20/238` [Viewed March 25, 2008].

[Pik17] Fedor G. Pikus. Read, copy, update... Then what?, September 2017. `https://www.youtube.com/watch?v=rxQ5K9lo034`.

[PMDY20] SeongJae Park, Paul E. McKenney, Laurent Dufour, and Heon Y. Yeom. An htm-based update-side synchronization for rcu on numa systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, Heraklion, Greece, 2020. Association for Computing Machinery.

[Pod10] Andrej Podzimek. Read-copy-update for opensolaris. Master's thesis, Charles University in Prague, 2010.

[Pok16] Michael Pokorny. The deadlock empire, February 2016. https://deadlockempire.github.io/.

[Pos08] PostgreSQL Global Development Group. PostgreSQL, November 2008. Available: https://www.postgresql.org/ [Viewed November 26, 2008].

[Pug90] William Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute of Advanced Computer Science Studies, Department of Computer Science, University of Maryland, College Park, Maryland, June 1990.

[Pug00] William Pugh. Reordering on an Alpha processor, 2000. Available: https://www.cs.umd.edu/~pugh/java/memoryModel/AlphaReordering.html [Viewed: June 23, 2004].

[Pul00] Geoffrey K. Pullum. How Dr. Seuss would prove the halting problem undecidable. *Mathematics Magazine*, 73(4):319–320, 2000. http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html.

[PW07] Donald E. Porter and Emmett Witchel. Lessons from large transactional systems, December 2007. Personal communication <20071214220521.GA5721@olive-green.cs.utexas.edu>.

[Ras14] Mindaugas Rasiukevicius. NPF—progress and perspective. In *AsiaBSDCon*, Tokyo, Japan, March 2014.

[Ras16] Mindaugas Rasiukevicius. Quiescent-state and epoch based reclamation, July 2016. https://github.com/rmind/libqsbr.

[Ray99] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 1999.

[RC15] Pedro Ramalhete and Andreia Correia. Poor man's URCU, August 2015. https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/poormanurcu-2015.pdf.

[RD12] Ravi Rajwar and Martin Dixon. Intel transactional synchronization extensions, September 2012. Intel Developer Forum (IDF) 2012 ARCS004.

[Reg10] John Regehr. A guide to undefined behavior in C and C++, part 1, July 2010. https://blog.regehr.org/archives/213.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly, Sebastopol, CA, USA, 2007.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, Austin, TX, December 2001. The Institute of Electrical and Electronics Engineers, Inc.

[RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Austin, TX, October 2002.

www.dbooks.org

[RH02] Zoran Radović and Erik Hagersten. Efficient synchronization for nonuniform communication architectures. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–13, Baltimore, Maryland, USA, November 2002. The Institute of Electrical and Electronics Engineers, Inc.

[RH03] Zoran Radović and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 241–252, Anaheim, California, USA, February 2003.

[RH18] Geoff Romer and Andrew Hunter. An RAII interface for deferred reclamation, March 2018. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0561r4.html`.

[RHP+07] Chistopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bhandari, and Emmett Witchel. TxLinux: Using and managing hardware transactional memory in an operating system. In *SOSP'07: Twenty-First ACM Symposium on Operating Systems Principles*, Stevenson, WA, USA, October 2007. ACM SIGOPS. Available: `http://www.sosp2007.org/papers/sosp056-rossbach.pdf` [Viewed October 21, 2007].

[Rin13] Martin Rinard. Parallel synchronization-free approximate data structure construction. In *Proceedings of the 5th USENIX Conference on Hot Topics in Parallelism*, HotPar'13, page 6, San Jose, CA, 2013. USENIX Association.

[RKM+10] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. *SIGARCH Comput. Archit. News*, 38(1):65–76, 2010.

[RLPB18] Yuxin Ren, Guyue Liu, Gabriel Parmer, and Björn Brandenburg. Scalable memory reclamation for multi-core, real-time systems. In *Proceedings of the 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 12, Porto, Portugal, April 2018. IEEE.

[RMF19] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The real-time Linux kernel: A survey on PREEMPT_RT. *ACM Comput. Surv.*, 52(1):18:1–18:36, February 2019.

[Ros06] Steven Rostedt. Lightweight PI-futexes, June 2006. Available: `https://www.kernel.org/doc/html/latest/locking/pi-futex.html` [Viewed February 14, 2021].

[Ros10a] Steven Rostedt. tracing: Harry Potter and the Deathly Macros, December 2010. Available: `https://lwn.net/Articles/418710/` [Viewed: August 28, 2011].

[Ros10b] Steven Rostedt. Using the TRACE_EVENT() macro (part 1), March 2010. Available: `https://lwn.net/Articles/379903/` [Viewed: August 28, 2011].

[Ros10c] Steven Rostedt. Using the TRACE_EVENT() macro (part 2), March 2010. Available: `https://lwn.net/Articles/381064/` [Viewed: August 28, 2011].

[Ros10d] Steven Rostedt. Using the TRACE_EVENT() macro (part 3), April 2010. Available: `https://lwn.net/Articles/383362/` [Viewed: August 28, 2011].

[Ros11] Steven Rostedt. lockdep: How to read its cryptic output, September 2011. http://www.linuxplumbersconf.org/2011/ocw/sessions/153.

[Roy17] Lance Roy. rcutorture: Add CBMC-based formal verification for SRCU, January 2017. URL: https://www.spinics.net/lists/kernel/msg2421833.html.

[RR20] Sergio Rajsbaum and Michel Raynal. Mastering concurrent computing through sequential thinking. *Commun. ACM*, 63(1):78–87, January 2020.

[RSB+97] Rajeev Rastogi, S. Seshadri, Philip Bohannon, Dennis W. Leinbaugh, Abraham Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 86–95, San Francisco, CA, USA, August 1997. Morgan Kaufmann Publishers Inc.

[RTY+87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *2nd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, Palo Alto, CA, October 1987. Association for Computing Machinery.

[Rus00a] Rusty Russell. Re: modular net drivers, June 2000. URL: http://oss.sgi.com/projects/netdev/archive/2000-06/msg00250.html [broken, February 15, 2021].

[Rus00b] Rusty Russell. Re: modular net drivers, June 2000. URL: http://oss.sgi.com/projects/netdev/archive/2000-06/msg00254.html [broken, February 15, 2021].

[Rus03] Rusty Russell. Hanging out with smart people: or... things I learned being a kernel monkey, July 2003. 2003 Ottawa Linux Symposium Keynote https://ozlabs.org/~rusty/ols-2003-keynote/ols-keynote-2003.html.

[Rut17] Mark Rutland. compiler.h: Remove ACCESS_ONCE(), November 2017. Git commit: https://git.kernel.org/linus/b899a850431e.

[SAE+18] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58âĂŞ66, March 2018.

[SAH+03] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Robert W. Wisniewski, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 141–154, San Antonio, Texas, USA, June 2003. USENIX Association.

[SATG+09] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards transactional memory semantics for C++. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 49–58, Calgary, AB, Canada, 2009. ACM.

[SBN+20] Dimitrios Siakavaras, Panagiotis Billis, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Efficient concurrent range queries in b+-trees using rcu-htm. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, page 571âĂŞ573, Virtual Event, USA, 2020. Association for Computing Machinery.

www.dbooks.org

[SBV10] Martin Schoeberl, Florian Brandner, and Jan Vitek. RTTM: Real-time transactional memory. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 326–333, 01 2010.

[Sch35] E. Schrödinger. Die gegenwärtige Situation in der Quantenmechanik. *Naturwissenschaften*, 23:807–812; 823–828; 844–849, November 1935.

[Sch94] Curt Schimmel. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[Sco06] Michael Scott. *Programming Language Pragmatics*. Morgan Kaufmann, Burlington, MA, USA, 2006.

[Sco13] Michael L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool, San Rafael, CA, USA, 2013.

[Sco15] Michael Scott. *Programming Language Pragmatics, 4th Edition*. Morgan Kaufmann, Burlington, MA, USA, 2015.

[Seq88] Sequent Computer Systems, Inc. *Guide to Parallel Programming*, 1988.

[Sew] Peter Sewell. Relaxed-memory concurrency. Available: `https://www.cl.cam.ac.uk/~pes20/weakmemory/` [Viewed: February 15, 2021].

[Sey12] Justin Seyster. *Runtime Verification of Kernel-Level Concurrency Using Compiler-Based Instrumentation*. PhD thesis, Stony Brook University, 2012.

[SF95] Janice M. Stone and Robert P. Fitzgerald. Storage in the PowerPC. *IEEE Micro*, 15(2):50–58, April 1995.

[Sha11] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, March 2011.

[She06] Gautham R. Shenoy. [patch 4/5] lock_cpu_hotplug: Redesign - lightweight implementation of lock_cpu_hotplug, October 2006. Available: `https://lkml.org/lkml/2006/10/26/73` [Viewed January 26, 2009].

[SHW11] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2011.

[Slo10] Lubos Slovak. First steps for utilizing userspace RCU library, July 2010. `https://gitlab.labs.nic.cz/knot/knot-dns/commit/f67acc0178ee9a781d7a63fb041b5d09eb5fb4a2`.

[SM95] John D. Slingwine and Paul E. McKenney. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Technical Report US Patent 5,442,758, Assigned to International Business Machines Corp, Washington, DC, August 1995.

[SM97] John D. Slingwine and Paul E. McKenney. Method for maintaining data coherency using thread activity summaries in a multicomputer system. Technical Report US Patent 5,608,893, Assigned to International Business Machines Corp, Washington, DC, March 1997.

[SM98] John D. Slingwine and Paul E. McKenney. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. Technical Report US Patent 5,727,209, Assigned to International Business Machines Corp, Washington, DC, March 1998.

[SM04a] Dipankar Sarma and Paul E. McKenney. Issues with selected scalability features of the 2.6 kernel. In *Ottawa Linux Symposium*, page 16, July 2004. https://www.kernel.org/doc/ols/2004/ols2004v2-pages-195-208.pdf.

[SM04b] Dipankar Sarma and Paul E. McKenney. Making RCU safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)*, pages 182–191, Boston, MA, USA, June 2004. USENIX Association.

[SM13] Thomas Sewell and Toby Murray. Above and beyond: seL4 noninterference and binary verification, May 2013. https://cps-vo.org/node/7706.

[Smi19] Richard Smith. Working draft, standard for programming language C++, January 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/n4800.pdf.

[SMS08] Michael Spear, Maged Michael, and Michael Scott. Inevitability mechanisms for software transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, Utah, February 2008. ACM. Available: http://www.cs.rochester.edu/u/scott/papers/2008_TRANSACT_inevitability.pdf [Viewed January 10, 2009].

[SNGK17] Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Combining HTM and RCU to implement highly efficient balanced binary search trees. In *12th ACM SIGPLAN Workshop on Transactional Computing*, Austin, TX, USA, February 2017.

[SPA94] SPARC International. *The SPARC Architecture Manual*, 1994. Available: https://sparc.org/wp-content/uploads/2014/01/SPARCV9.pdf.gz.

[Spi77] Keith R. Spitz. Tell which is which and you'll be rich, 1977. Inscription on wall of dungeon.

[Spr01] Manfred Spraul. Re: RFC: patch to allow lock-free traversal of lists with insertion, October 2001. URL: http://lkml.iu.edu/hypermail/linux/kernel/0110.1/0410.html.

[Spr08] Manfred Spraul. [RFC, PATCH] state machine based rcu, August 2008. Available: https://lkml.org/lkml/2008/8/21/336 [Viewed December 8, 2008].

[SR84] Z. Segall and L. Rudolf. Dynamic decentralized cache schemes for MIMD parallel processors. In *11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[SRK+11] Justin Seyster, Prabakar Radhakrishnan, Samriti Katoch, Abhinav Duggal, Scott D. Stoller, and Erez Zadok. Redflag: a framework for analysis of kernel-level concurrency. In *Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I*, ICA3PP'11, pages 66–79, Melbourne, Australia, 2011. Springer-Verlag.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[SS94] Duane Szafron and Jonathan Schaeffer. Experimentally assessing the usability of parallel programming systems. In *IFIP WG10.3 Programming Environments*

www.dbooks.org

for Massively Parallel Distributed Systems, pages 19.1–19.7, Monte Verita, Ascona, Switzerland, 1994.

[SS06] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. POWER and ARM litmus tests, 2011. `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test6.pdf`.

[SSHT93] Janice S. Stone, Harold S. Stone, Philip Heidelberger, and John Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology Systems and Applications*, 1(4):58–71, November 1993.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, Chichester, West Sussex, England, 2000.

[SSVM02] S. Swaminathan, John Stultz, Jack Vogel, and Paul E. McKenney. Fairlocks – a high performance fair locking scheme. In *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 246–251, Cambridge, MA, USA, November 2002.

[ST87] William E. Snaman and David W. Thiel. The VAX/VMS distributed lock manager. *Digital Technical Journal*, 5:29–44, September 1987.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, Ottawa, Ontario, Canada, August 1995.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, 1992.

[Ste13] W. Richard Stevens. *Advanced Programming in the UNIX Environment, 3rd Edition*. Addison Wesley, 2013.

[Sut08] Herb Sutter. Effective concurrency, 2008. Series in Dr. Dobbs Journal.

[Sut13] Adrian Sutton. Concurrent programming with the Disruptor, January 2013. Presentation at Linux.conf.au 2013, URL: `https://www.youtube.com/watch?v=ItpT_vmRHyI`.

[SW95] Richard L. Sites and Richard T. Witek. *Alpha AXP Architecture*. Digital Press, second edition, 1995.

[SWS16] Harshal Sheth, Aashish Welling, and Nihar Sheth. Read-copy update in a garbage collected environment, 2016. MIT PRIMES program: `https://math.mit.edu/research/highschool/primes/materials/2016/conf/10-1%20Sheth-Welling-Sheth.pdf`.

[SZJ12] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. Eliminating read barriers through procrastination and cleanliness. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 49–60, Beijing, China, 2012. ACM.

[Tal07] Nassim Nicholas Taleb. *The Black Swan*. Random House, 2007.

[TDV15] Joseph Tassarotti, Derek Dreyer, and Victor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *Proceedings of the 2015 Proceedings of the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '15, pages 110–120, New York, NY, USA, June 2015. ACM.

[The08] The Open MPI Project. Open MPI, November 2008. Available: http://www.open-mpi.org/software/ [Viewed November 26, 2008].

[The11] The Valgrind Developers. Valgrind, November 2011. http://www.valgrind.org/.

[The12a] The NetBSD Foundation. pserialize(9), October 2012. http://netbsd.gw.com/cgi-bin/man-cgi?pserialize+9+NetBSD-current.

[The12b] The OProfile Developers. Oprofile, April 2012. http://oprofile.sourceforge.net.

[TMW11] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX Annual Technical Conference*, pages 145–158, Portland, OR USA, June 2011. The USENIX Association.

[Tor01] Linus Torvalds. Re: [Lse-tech] Re: RFC: patch to allow lock-free traversal of lists with insertion, October 2001. URL: https://lkml.org/lkml/2001/10/13/105, https://lkml.org/lkml/2001/10/13/82.

[Tor02] Linus Torvalds. Linux 2.5.43, October 2002. Available: https://lkml.org/lkml/2002/10/15/425 [Viewed March 30, 2008].

[Tor03] Linus Torvalds. Linux 2.6, August 2003. Available: https://kernel.org/pub/linux/kernel/v2.6 [Viewed February 16, 2021].

[Tor08] Linus Torvalds. Move ACCESS_ONCE() to <linux/compiler.h>, May 2008. Git commit: https://git.kernel.org/linus/9c3cdc1f83a6.

[Tor19] Linus Torvalds. rcu: locking and unlocking need to always be at least barriers, June 2019. Git commit: https://git.kernel.org/linus/66be4e66a7f4.

[Tra01] Transaction Processing Performance Council. TPC, 2001. Available: http://www.tpc.org/ [Viewed December 7, 2008].

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism, April 1986. RJ 5118.

[Tri12] Josh Triplett. *Relativistic Causal Ordering: A Memory Model for Scalable Concurrent Data Structures*. PhD thesis, Portland State University, 2012.

[TS93] Hiroaki Takada and Ken Sakamura. A bounded spin lock algorithm with preemption. Technical Report 93-02, University of Tokyo, Tokyo, Japan, 1993.

[TS95] H. Takada and K. Sakamura. Real-time scalability of nested spin locks. In *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, RTCSA '95, pages 160–167, Tokyo, Japan, 1995. IEEE Computer Society.

[Tur37] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42 of *2*, pages 230–265, 1937.

[TZK+13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, Farminton, Pennsylvania, 2013. ACM.

www.dbooks.org

[Ung11] David Ungar. Everything you know (about parallel programming) is wrong!: A wild screed about the future. In *Dynamic Languages Symposium 2011*, Portland, OR, USA, October 2011. Invited talk presentation.

[Uni08a] University of California, Berkeley. BOINC: compute for science, October 2008. Available: `http://boinc.berkeley.edu/` [Viewed January 31, 2008].

[Uni08b] University of California, Berkeley. SETI@HOME, December 2008. Available: `http://setiathome.berkeley.edu/` [Viewed January 31, 2008].

[Uni10] University of Maryland. Parallel maze solving, November 2010. URL: `http://www.cs.umd.edu/class/fall2010/cmsc433/p3/` [broken, February 2021].

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222, Ottowa, Ontario, Canada, 1995. ACM.

[Van18] Michal Vaner. ArcSwap, April 2018. `https://crates.io/crates/arc-swap`.

[VBC+15] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the c11 memory model and what we can do about it. *SIGPLAN Not.*, 50(1):209–220, January 2015.

[VGS08] Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, Utah, USA, February 2008. ACM. Available: `http://www.cs.wisc.edu/multifacet/papers/transact08_txlock.pdf` [Viewed September 7, 2009].

[Vog09] Werner Vogels. Eventually consistent. *Commun. ACM*, 52:40–44, January 2009.

[Š11] Jaroslav Ševčík. Safe optimisations for shared-memory concurrent programs. *SIGPLAN Not.*, 46(6):306–316, June 2011.

[Wav16] Wave Computing, Inc. *MIPS®Architecture For Programmers Volume II-A: The MIPS64®Instruction Set Reference Manual*, 2016. URL: `https://www.mips.com/downloads/the-mips64-instruction-set-v6-06/`.

[Wei63] J. Weizenbaum. Symmetric list processor. *Commun. ACM*, 6(9):524–536, September 1963.

[Wei12] Frédéric Weisbecker. Interruption timer périodique, 2012. `http://www.dailymotion.com/video/xtxtew_interruption-timer-periodique-frederic-weisbecker-kernel-recipes-12_tech`.

[Wei13] Stewart Weiss. Unix lecture notes, May 2013. Available: `http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/unix_lecture_notes/` [Viewed April 8, 2014].

[Wik08] Wikipedia. Zilog Z80, 2008. Available: `https://en.wikipedia.org/wiki/Z80` [Viewed: December 7, 2008].

[Wik12] Wikipedia. Labyrinth, January 2012. `https://en.wikipedia.org/wiki/Labyrinth`.

[Wil12] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning, Shelter Island, NY, USA, 2012.

[Wil19] Anthony Williams. *C++ Concurrency in Action, 2nd Edition*. Manning, Shelter Island, NY, USA, 2019.

[WKS94] Robert W. Wisniewski, Leonidas Kontothanassis, and Michael L. Scott. Scalable spin locks for multiprogrammed systems. In *8th IEEE Int'l. Parallel Processing Symposium*, Cancun, Mexico, April 1994. The Institute of Electrical and Electronics Engineers, Inc.

[WTS96] Cai-Dong Wang, Hiroaki Takada, and Ken Sakamura. Priority inheritance spin locks for multiprocessor real-time systems. In *Proceedings of the 2nd International Symposium on Parallel Architectures, Algorithms, and Networks*, ISPAN '96, pages 70–76, Beijing, China, 1996. IEEE Computer Society.

[xen14] xenomai.org. Xenomai, December 2014. URL: `http://xenomai.org/`.

[Xu10] Herbert Xu. bridge: Add core IGMP snooping support, February 2010. Available: `https://marc.info/?t=126719855400006&r=1&w=2` [Viewed March 20, 2011].

[YHLR13] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel® Transactional Synchronization Extensions for high-performance computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, Denver, Colorado, 2013. ACM.

[Yod04a] Victor Yodaiken. Against priority inheritance, September 2004. Available: `https://www.yodaiken.com/papers/inherit.pdf` [Viewed May 26, 2007].

[Yod04b] Victor Yodaiken. Temporal inventory and real-time synchronization in RTLinuxPro, September 2004. URL: `https://www.yodaiken.com/papers/sync.pdf`.

[Zel11] Cyril Zeller. CUDA C/C++ basics: Supercomputing 2011 tutorial, November 2011. `https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf`.

[Zha89] Lixia Zhang. *A New Architecture for Packet Switching Network Protocols*. PhD thesis, Massachusetts Institute of Technology, July 1989.

[Zij14] Peter Zijlstra. Another go at speculative page faults, October 2014. `https://lkml.org/lkml/2014/10/20/620`.

www.dbooks.org

Credits

If I have seen further it is by standing on the shoulders of giants.

Isaac Newton, modernized

LaTeX Advisor

Akira Yokosawa is this book's LaTeX advisor, which perhaps most notably includes the care and feeding of the style guide laid out in Appendix D. This work includes table layout, listings, fonts, rendering of math, acronyms, bibliography formatting, epigraphs, hyperlinks, paper size. Akira also perfected the cross-referencing of quick quizzes, allowing easy and exact navigation between quick quizzes and their answers. He also added build options that permit quick quizzes to be hidden and to be gathered at the end of each chapter, textbook style.

This role also includes the build system, which Akira has optimized and made much more user-friendly. His enhancements have included automating response to bibliography changes, automatically determining which source files are present, and automatically generating listings (with automatically generated hyperlinked line-number references) from the source files.

Reviewers

- Alan Stern (Chapter 15).
- Andy Whitcroft (Section 9.5.2, Section 9.5.3).
- Artem Bityutskiy (Chapter 15, Appendix C).
- Dave Keck (Appendix C).
- David S. Horner (Section 12.1.5).
- Gautham Shenoy (Section 9.5.2, Section 9.5.3).
- "jarkao2", AKA LWN guest #41960 (Section 9.5.3).
- Jonathan Walpole (Section 9.5.3).
- Josh Triplett (Chapter 12).
- Michael Factor (Section 17.2).
- Mike Fulton (Section 9.5.2).
- Peter Zijlstra (Section 9.5.4).
- Richard Woodruff (Appendix C).
- Suparna Bhattacharya (Chapter 12).
- Vara Prasad (Section 12.1.5).

Reviewers whose feedback took the extremely welcome form of a patch are credited in the git logs.

Machine Owners

Readers might have noticed some graphs showing scalability data out to several hundred CPUs, courtesy of my current employer, with special thanks to Paul Saab, Yashar Bayani, Joe Boyd, and Kyle McMartin.

From back in my time at IBM, a great debt of thanks goes to Martin Bligh, who originated the Advanced Build and Test (ABAT) system at IBM's Linux Technology Center, as well as to Andy Whitcroft, Dustin Kirkland, and many others who extended this system. Many thanks go also to a great number of machine owners: Andrew Theurer, Andy Whitcroft, Anton Blanchard, Chris McDermott, Cody Schaefer, Darrick Wong, David "Shaggy" Kleikamp, Jon M. Tollefson, Jose R. Santos, Marvin Heffler, Nathan Lynch, Nishanth Aravamudan, Tim Pepper, and Tony Breeds.

Original Publications

1. Section 2.4 ("What Makes Parallel Programming Hard?") on page 13 originally appeared in a Portland State University Technical Report [MGM+09].
2. Section 4.3.4.1 ("Shared-Variable Shenanigans") on page 40 originally appeared in Linux Weekly News [ADF+19].

www.dbooks.org

3. Section 6.5 (“Retrofitted Parallelism Considered Grossly Sub-Optimal”) on page 92 originally appeared in 4th USENIX Workshop on Hot Topics on Parallelism [McK12c].

4. Section 9.5.2 (“RCU Fundamentals”) on page 140 originally appeared in Linux Weekly News [MW07].

5. Section 9.5.3 (“RCU Linux-Kernel API”) on page 146 originally appeared in Linux Weekly News [McK08d].

6. Section 9.5.4 (“RCU Usage”) on page 155 originally appeared in Linux Weekly News [McK08f].

7. Section 9.5.5 (“RCU Related Work”) on page 166 originally appeared in Linux Weekly News [McK14e].

8. Section 9.5.5 (“RCU Related Work”) on page 166 originally appeared in Linux Weekly News [MP15a].

9. Chapter 12 (“Formal Verification”) on page 219 originally appeared in Linux Weekly News [McK07f, MR08, McK11d].

10. Section 12.3 (“Axiomatic Approaches”) on page 249 originally appeared in Linux Weekly News [MS14].

11. Section 13.5.4 (“Correlated Fields”) on page 268 originally appeared in Oregon Graduate Institute [McK04].

12. Chapter 15 (“Advanced Synchronization: Memory Ordering”) on page 297 originally appeared in the Linux kernel [HMDZ06].

13. Chapter 15 (“Advanced Synchronization: Memory Ordering”) on page 297 originally appeared in Linux Weekly News [AMM+17a, AMM+17b].

14. Chapter 15 (“Advanced Synchronization: Memory Ordering”) on page 297 originally appeared in ASPLOS ’18 [AMM+18].

15. Section 15.3.2 (“Address- and Data-Dependency Difficulties”) on page 318 originally appeared in the Linux kernel [McK14c].

16. Section 15.5 (“Memory-Barrier Instructions For Specific CPUs”) on page 331 originally appeared in Linux Journal [McK05a, McK05b].

Figure Credits

1. Figure 3.1 (p 17) by Melissa Broussard.
2. Figure 3.2 (p 18) by Melissa Broussard.
3. Figure 3.3 (p 18) by Melissa Broussard.
4. Figure 3.5 (p 19) by Melissa Broussard.
5. Figure 3.6 (p 20) by Melissa Broussard.
6. Figure 3.7 (p 20) by Melissa Broussard.
7. Figure 3.8 (p 21) by Melissa Broussard.
8. Figure 3.9 (p 21) by Melissa Broussard.
9. Figure 3.11 (p 25) by Melissa Broussard.
10. Figure 5.3 (p 51) by Melissa Broussard.
11. Figure 6.1 (p 73) by Kornilios Kourtis.
12. Figure 6.2 (p 74) by Melissa Broussard.
13. Figure 6.3 (p 74) by Kornilios Kourtis.
14. Figure 6.4 (p 75) by Kornilios Kourtis.
15. Figure 6.13 (p 84) by Melissa Broussard.
16. Figure 6.14 (p 85) by Melissa Broussard.
17. Figure 6.15 (p 85) by Melissa Broussard.
18. Figure 7.1 (p 100) by Melissa Broussard.
19. Figure 7.2 (p 100) by Melissa Broussard.
20. Figure 10.13 (p 183) by Melissa Broussard.
21. Figure 10.14 (p 184) by Melissa Broussard.
22. Figure 11.1 (p 199) by Melissa Broussard.
23. Figure 11.2 (p 200) by Melissa Broussard.
24. Figure 11.3 (p 205) by Melissa Broussard.
25. Figure 11.6 (p 217) by Melissa Broussard.
26. Figure 14.1 (p 277) by Melissa Broussard.
27. Figure 14.2 (p 277) by Melissa Broussard.
28. Figure 14.3 (p 278) by Melissa Broussard.
29. Figure 14.10 (p 286) by Melissa Broussard.

30. Figure 14.11 (p 286) by Melissa Broussard.
31. Figure 14.14 (p 288) by Melissa Broussard.
32. Figure 14.15 (p 295) by Sarah McKenney.
33. Figure 14.16 (p 295) by Sarah McKenney.
34. Figure 15.2 (p 299) by Melissa Broussard.
35. Figure 15.5 (p 305) by Akira Yokosawa.
36. Figure 15.18 (p 336) by Melissa Brossard.
37. Figure 16.2 (p 345) by Melissa Broussard.
38. Figure 17.1 (p 347) by Melissa Broussard.
39. Figure 17.2 (p 348) by Melissa Broussard.
40. Figure 17.3 (p 348) by Melissa Broussard.
41. Figure 17.4 (p 349) by Melissa Broussard.
42. Figure 17.5 (p 349) by Melissa Broussard, remixed.
43. Figure 17.9 (p 362) by Melissa Broussard.
44. Figure 17.10 (p 362) by Melissa Broussard.
45. Figure 17.11 (p 362) by Melissa Broussard.
46. Figure 17.12 (p 363) by Melissa Broussard.
47. Figure 18.1 (p 385) by Melissa Broussard.
48. Figure 18.1 (p 385) by Melissa Broussard.
49. Figure A.2 (p 390) by Melissa Broussard.
50. Figure E.2 (p 464) by Kornilios Kourtis.

Figure 9.30 was adapted from Fedor Pikus's "When to use RCU" slide [Pik17]. The discussion of mechanical reference counters in Section 9.2 stemmed from a private conversation with Dave Regan.

Other Support

We owe thanks to many CPU architects for patiently explaining the instruction- and memory-reordering features of their CPUs, particularly Wayne Cardoza, Ed Silha, Anton Blanchard, Tim Slegel, Juergen Probst, Ingo Adlung, Ravi Arimilli, Cathy May, Derek Williams, H. Peter Anvin, Andy Glew, Leonid Yegoshin, Richard Grisenthwaite, and Will Deacon. Wayne deserves special thanks for his patience in explaining Alpha's reordering of dependent loads, a lesson that Paul resisted quite strenuously!

The bibtex-generation service of the Association for Computing Machinery has saved us a huge amount of time and effort compiling the bibliography, for which we are grateful. Thanks are also due to Stamatis Karnouskos, who convinced me to drag my antique bibliography database kicking and screaming into the 21st century. Any technical work of this sort owes thanks to the many individuals and organizations that keep Internet and the World Wide Web up and running, and this one is no exception.

Portions of this material are based upon work supported by the National Science Foundation under Grant No. CNS-0719851.

www.dbooks.org