The Design and Implementation of the Anykernel and Rump Kernels

Antti Kantee

2nd Edition

www.dbooks.org

Copyright (c) 2012, 2014, 2015, 2016 Antti Kantee <pooka@iki.fi>
All Rights Reserved.

You may download this book for study from `http://book.rumpkernel.org/` in eletronic format free of charge. You MAY NOT redistribute this book or its contents in electronic format.

You may distribute hardcopies of the book with or without fee, provided that the following conditions are met. The hardcopy must be produced by printing the pdf from `http://book.rumpkernel.org/`. Doublesided printing must be used, and the book must be bound with even-numbered pages on the left and the odd-numbered pages on the right. Absolutely no modifications may be made, including the addition or removal of text, or the addition or removal of full pages or covers. Each hardcopy must be bound as an individual unit, and not included for example in a collection of works.

Exceptions to these terms require prior agreement in writing.

Build information

```
Revision: 7eda996413beed925d1df3fed6f327e709ae4119
Date:     20160802
```

www.dbooks.org

Preface

I'll tip my hat to the new constitution
Take a bow for the new revolution
Smile and grin at the change all around
Pick up my guitar and play
Just like yesterday
Then I'll get on my knees and pray
We don't get fooled again
– The Who

This document is intended as an up-to-date description on the fundamental concepts related to the anykernel and rump kernels. It is based on the dissertation written in 2011 and early 2012: *Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels*.

The major change with rump kernels since the first edition is a shift in focus and motivation. In work leading up to the first edition, rump kernels were about running kernel components in userspace. That work defined the core architecture, and that definition is still valid and accurate. Since then, work has focused on harnessing the potential of rump kernels for building entirely new computation stacks.

Since this edition of the book is no longer an academic document, we do not support every statement we make with a citation or experiment. In fact, we also take the liberty to present opinions which are open for debate.

www.dbooks.org

Acknowledgments

Note: these acknowledgments are specifically about contributions to this book, not to the underlying subject of discussion. Given that the main purpose of the described work is to reuse decades worth of existing code, listing all contributers to the code is a herculean task. Should you be truly interested in who contributed what, the provenance is available from public repository logs.

This book is based on the first edition. The people who proofread the first edition and provided detailed improvement suggestions are Dr. Marshall Kirk McKusick, Dr. Thomas Klausner and Dr. Xi Zhang. Additional thanks in the form of formal credits for the first edition go to Prof. Heikki Saikkonen, Prof. Renzo Davoli and Dr. Peter Tröger.

The cover art was contributed by Sebastian Sala. It is based on the rump kernel project logo, the author of which Sebastian also is.

Parts of the Introduction and Conclusions first appeared in the author's article "The Rise and Fall of the Operating System" in USENIX ;login: Vol. 40, No. 5 (October 2015). Rik Farrow provided multiple rounds of feedback and suggestions on the article.

Donations

The following individuals and corporations provided donations to support work on the second edition of this book. Thank you!

Ionut Hristodorescu / NettoLogic

Danny Fullerton

Anonymous

Mehdi Ahmadi

Neeraj Sharma

www.dbooks.org

Contents

Preface **5**

Contents **9**

List of Abbreviations **13**

List of Figures **17**

List of Tables **19**

1 Introduction **21**

1.1 Operating Systems 21

1.1.1 Historical Perspective 22

1.1.2 And Where It Got Us 23

1.1.3 What We Really Need 25

1.2 The Anykernel and Rump Kernels 26

1.3 Book Outline 29

1.4 Further Material 30

1.4.1 Source Code 30

1.4.2 Manual Pages 31

2 Concepts: Anykernel and Rump Kernels **33**

2.1 Driving Drivers 35

2.1.1 Relegation and Reuse 35

2.1.2 Base, Orthogonal Factions, Drivers 36

2.1.3 Hosting 39

2.2 Rump Kernel Clients 39

2.3 Threads and Schedulers 45

2.3.1 Kernel threads 49

2.3.2 A CPU for a Thread 50

www.dbooks.org

2.3.3 Interrupts and Preemption 52
2.3.4 An Example 53
2.4 Virtual Memory 54
2.5 Distributed Services with Remote Clients 56
2.6 Summary 57

3 Implementation: Anykernel and Rump Kernels 59
3.1 Kernel Partitioning 59
3.1.1 Extracting and Implementing 63
3.1.2 Providing Components 64
3.2 Running the Kernel in an Hosted Environment 66
3.2.1 C Symbol Namespaces 66
3.2.2 Privileged Instructions 69
3.2.3 The Hypercall Interface(s) 69
3.3 Rump Kernel Entry and Exit 73
3.3.1 CPU Scheduling 76
3.3.2 Interrupts and Soft Interrupts 83
3.4 Virtual Memory Subsystem 85
3.4.1 Page Remapping 87
3.4.2 Memory Allocators 90
3.4.3 Pagedaemon 91
3.5 Synchronization 94
3.5.1 Passive Serialization Techniques 96
3.5.2 Spinlocks on a Uniprocessor Rump Kernel 101
3.6 Application Interfaces to the Rump Kernel 103
3.6.1 System Calls 104
3.6.2 vnode Interface 109
3.6.3 Interfaces Specific to Rump Kernels 111
3.7 Rump Kernel Root File System 112
3.7.1 Extra-Terrestrial File System 113

3.7.2 External Storage . . . 114
3.8 Attaching Components . . . 115
3.8.1 Kernel Modules . . . 115
3.8.2 Modules: Supporting Standard Binaries . . . 119
3.8.3 Rump Component Init Routines . . . 123
3.9 I/O Backends . . . 124
3.9.1 Networking . . . 127
3.9.2 Disk Driver . . . 134
3.10 Hardware Devices: A Case of USB . . . 136
3.10.1 Conclusions of the USB Approach . . . 137
3.11 Microkernel Servers: Case Study with File Servers . . . 138
3.11.1 Mount Utilities and File Servers . . . 138
3.11.2 Requests: The p2k Library . . . 141
3.11.3 Unmounting . . . 142
3.11.4 Security Benefits . . . 143
3.12 Remote Clients . . . 144
3.12.1 Client-Kernel Locators . . . 146
3.12.2 The Client . . . 146
3.12.3 The Server . . . 147
3.12.4 Communication Protocol . . . 149
3.12.5 Of Processes and Inheritance . . . 154
3.12.6 Host System Call Hijacking . . . 155
3.12.7 A Tale of Two Syscalls: `fork()` and `execve()` . . . 160
3.12.8 Performance . . . 163
3.13 Experiment: Bootstrap Time . . . 165
3.14 Summary . . . 168

4 Rump Kernel Ecosystem **171**
4.1 buildrump.sh . . . 171
4.1.1 The Double-crossing Toolchain . . . 172

www.dbooks.org

4.1.2 POSIX Host Hypercalls . . . 173
4.1.3 Full Userspace Build . . . 174
4.1.4 src-netbsd . . . 174
4.2 Rumprun Unikernel . . . 176
4.2.1 bmk – Bare Metal Kernel . . . 177
4.2.2 Rumpuser . . . 181
4.2.3 To Userspace (or Not To Userspace) . . . 182
4.2.4 Toolchain . . . 186
4.2.5 PCI: Big Yellow Bus . . . 191
4.3 rumpctrl . . . 193
4.4 fs-utils . . . 195
4.5 Summary . . . 196

5 Short History 197
5.1 First Steps . . . 197
5.2 Towards Robust and Maintainable . . . 199
5.3 Syscall Support . . . 200
5.4 Beyond File Systems . . . 201
5.5 Symbol Isolation . . . 202
5.6 Local Clients . . . 203
5.7 Remote Clients . . . 204
5.8 Shifting Focus Towards Driver Reuse . . . 204
5.9 We're not in Kansas Anymore . . . 205
5.10 Summary . . . 208

6 Conclusions 209

References 211

List of Abbreviations

ABI	Application Binary Interface: The interface between binaries. ABI-compatible binaries can interface with each other.
ASIC	Application-Specific Integrated Circuit (or Cloud ;-)
CAS	Compare-And-Swap; atomic operation
CPU	Central Processing Unit; in this document the term is context-dependant. It is used to denote both the physical hardware unit or a virtual concept of a CPU.
DMA	Direct Memory Access
DSL	Domain Specific Language
DSO	Dynamic Shared Object
ELF	Executable and Linking Format; the binary format used by NetBSD and some other modern Unix-style operating systems.
FFS	Berkeley Fast File System; in most contexts, this can generally be understood to mean the same as UFS (Unix File System).
FS	File System
GPL	General Public License; a software license
HTTP	HyperText Transfer Protocol
i386	Intel 32-bit ISA (a.k.a. IA-32)

www.dbooks.org

IPC	Inter-Process Communication
ISA	Instruction Set Architecture
LGPL	Lesser GPL; a less restrictive variant of GPL
LRU	Least Recently Used
LWP	Light Weight Process; the kernel's idea of a thread. This acronym is usually written in lowercase (lwp) to mimic the kernel structure name (`struct lwp`).
MD	Machine Dependent [code]; [code] specific to the platform
MI	Machine Independent [code]; [code] usable on all platforms
MMU	Memory Management Unit: hardware unit which handles memory access and does virtual-to-physical translation for memory addresses
NIC	Network Interface Controller
OS	Operating System
OS	Orchestrating System
PCI	Peripheral Component Interconnect; hardware bus
PIC	Position Independent Code; code which uses relative addressing and can be loaded at any location. It is typically used in shared libraries, where the load address of the code can vary from one process to another.
PR	Problem Report
RTT	RoundTrip Time

RUMP	Deprecated “backronym” denoting a rump kernel and its local client application. This backronym should not appear in any material written since mid-2010.
SLIP	Serial Line IP: protocol for framing IP datagrams over a serial line.
TLS	Thread-Local Storage; private per-thread data
TLS	Transport Layer Security
UML	User Mode Linux
USB	Universal Serial Bus
VAS	Virtual Address Space
VM	Virtual Memory; the abbreviation is context dependent and can mean both virtual memory and the kernel’s virtual memory subsystem
VMM	Virtual Machine Monitor

www.dbooks.org

List of Figures

1.1 Relationship between key concepts 28

2.1 Rump kernel hierarchy 37
2.2 BPF access via file system 40
2.3 BPF access without a file system 41
2.4 Client types illustrated 42
2.5 Use of `curcpu()` in the pool allocator 51
2.6 Providing memory mapping support on top of a rump kernel 55

3.1 Performance of position independent code (PIC) 65
3.2 C namespace protection 67
3.3 Rump kernel entry/exit pseudocode 75
3.4 System call performance using the trivial CPU scheduler 77
3.5 CPU scheduling algorithm in pseudocode 79
3.6 CPU release algorithm in pseudocode 81
3.7 System call performance using the improved CPU scheduler 82
3.8 Performance of page remapping vs. copying 89
3.9 Using CPU cross calls when checking for syscall users 99
3.10 Cost of atomic memory bus locks on a twin core host 102
3.11 Call stub for `rump_sys_lseek()` 106
3.12 Compile-time optimized `sizeof()` check 108
3.13 Implementation of `RUMP_VOP_READ()` 110
3.14 Application interface implementation of lwproc `rfork()` 111
3.15 Comparison of `pmap_is_modified` definitions 121
3.16 Comparison of `curlwp` definitions 122
3.17 Example: selected contents of `netinet_component.c` 126
3.18 Networking options for rump kernels 127
3.19 Bridging a tap interface to the host's `re0` 129

www.dbooks.org

3.20 sockin attachment 133
3.21 File system server 139
3.22 Use of `-o rump` in `/etc/fstab` 140
3.23 Implementation of `p2k_node_read()` 142
3.24 Mounting a corrupt FAT FS with the kernel driver in a rump kernel 143
3.25 Remote client architecture 145
3.26 Example invocations for `rump_server` 148
3.27 System call hijacking 156
3.28 Implementation of `fork()` on the client side 161
3.29 Local vs. Remote system call overhead 163
3.30 Time required to bootstrap one rump kernel 165
3.31 Script for starting, configuring and testing a network cluster 167
3.32 Network cluster startup time 168

4.1 Rumprun software stack 178
4.2 Rumprun stack trace for `sleep()` 181
4.3 Userspace aliases for rump kernel syscalls 183
4.4 Building a runnable Rumprun unikernel image 189
4.5 Architecture of rumpctrl 194

List of Tables

2.1 Comparison of client types 43

3.1 Symbol renaming illustrated 68
3.2 File system I/O performance vs. available memory 93
3.3 Kernel module classification 116
3.4 Component classes 125
3.5 Requests from the client to the kernel 150
3.6 Requests from the kernel to the client 151
3.7 Step-by-step comparison of host and rump kernel syscalls, part 1/2 152
3.8 Step-by-step comparison of host and rump kernel syscalls, part 2/2 153

4.1 src-netbsd branches 175

www.dbooks.org

1 Introduction

The mission of the first edition of this book (2012) was to introduce the anykernel and rump kernels and motivate their existence. Additionally, we explored the characteristics of the technology through various experiments. The paramount, often criminally overlooked experiment was the one hiding in plain sight: is it possible to construct the system in a sustainable, real-world compatible fashion. That paramount experiment was shown to be a success, and that result has not changed since the original publication, only strengthened. The core technology is still almost identical to the one described in the original book.

This new edition has been written to account for the practical experiences from new use cases, many of which were proposed in the first edition, but which have since become reality.

To start off, we will look at operating systems in general: what one is, how they developed throughout history, where they are now, what the problem is, and why the time is now ripe for change. After that, we will briefly introduce the Anykernel and Rump Kernels, and argue why they are part of the solution to the problem.

1.1 Operating Systems

The term *operating system* originally meant a system which aids computer operators in loading tapes and punchcards onto the computer [15]. We take a slightly more modern approach, and define an operating system as a collection of subroutines which allow application programs to run on a given *platform*. The platform can be for example a physical unit of hardware, or be virtually provisioned such as on the cloud. Additionally, an operating system may, for example, multiplex the platform

www.dbooks.org

for a number of applications, protect applications from each other, be distributed in nature, or provide an interface which is visually appealing to some people.

The majority of the operating system is made up of *drivers*, which abstract some underlying entity. For example, device drivers know which device registers to read and write for the desired result, file system drivers know which blocks contain which parts of which files, and so forth. In essence, a driver is a *protocol translator*, which transforms requests and responses to different representations.

There is nothing about protocol translation which dictates that a driver must be an integral part of an operating system as opposed to being part of application code. However, operating system drivers may also be used as a tool for imposing protection boundaries. For example, an operating system may require that applications access a storage device through the file system driver. The file system can then enforce that users are reading or writing only the storage blocks that they are supposed to have access to. As we shall see shortly, imposing privilege boundaries grew out of historic necessity when computers were few and the operating system was a tool to multiplex a single machine for many users.

1.1.1 Historical Perspective

Computers were expensive in the 1950's and 1960's. For example, the cost of the UNIVAC I in 1951 was just short of a million dollars. Accounting for inflation, that is approximately 9 million dollars in today's money. Since it was desirable to keep expensive machines doing something besides idling, batch scheduling was used to feed new computations and keep idletime to a minimum.

As most of us intuitively know, reaching the solution of a problem is easier if you are allowed to stumble around with constant feedback, as compared to a situation where

you must have holistic clairvoyance over the entire scenario before you even start. The lack of near-instant feedback was a problem with batch systems. You submitted a job, context switched to something else, came back the next day, context switched back to your computation, and discovered your program was missing a comma.

To address the feedback problem, timesharing was invented. Users logged into a machine via a terminal and got the illusion of having the whole system to themselves. The timesharing operating system juggled between users and programs. Thereby, poetic justice was administered: the computer was now the one context-switching, not the human. Going from running one program at a time to running multiple at the "same" time required more complex control infrastructure. The system had to deal with issues such as hauling programs in and out of memory depending on if they were running or not (swapping), scheduling the tasks according to some notion of fairness, and providing users with private, permanent storage (file system). In other words, 50 years ago they had the key concepts of current operating systems figured out. What has happened since?

1.1.2 And Where It Got Us

The early timesharing systems isolated users from other users. The average general purpose operating system still does a decent job at isolating users from each other. However, that type of isolation does little good in a world which does not revolve around people logging into a timesharing system. The increasing problem is isolating the user from herself or himself. Ages ago, when you yourself wrote all of the programs you ran, or at least had a physical interaction possibility with the people who did, you could be reasonably certain that a program you ran did not try to steal your credit card numbers. These days, when you download a million lines of so-so trusted application code from the Internet, you have no idea of what happens when you run it on a traditional operating system.

www.dbooks.org

The timesharing system also isolates the system and hardware components from the unprivileged user. In this age when everyone has their own hardware — virtual if not physical — that isolation vector is of questionable value. It is no longer a catastrophe if an unprivileged process binds to transport layer ports less than 1024. Everyone should consider reading and writing the network medium unlimited due to hardware no longer costing a million, regardless of what the operating system on some system does. The case for separate system and user software components is therefore no longer universal. Furthermore, the abstract interfaces which hide underlying power, especially that of modern I/O hardware, are insufficient for high-performance computing [45].

In other words, since the operating system does not protect the user from evil or provide powerful abstractions, it fails its mission in the modern world. Why do we keep on using such systems? Let us imagine the world of computing as a shape sorter. In the beginning, all holes were square: all computation was done on a million dollar machine sitting inside of a mountain. Square pegs were devised to fit the square holes, as one would logically expect. The advent of timesharing brought better square pegs, but it did so in the confines of the old scenario of the mountain-machine. Then the world of computing diversified. We got personal computing, we got mobile devices, we got IoT, we got the cloud. Suddenly, we had round holes, triangular holes and the occasional trapezoid and rhombus. Yet, we are still fascinated by square-shaped pegs, and desperately try to cram them into every hole, regardless of if they fit or not.

Why are we so fascinated with square-shaped pegs? What happens if we throw away the entire operating system? The first problem with that approach is, and it is a literal show-stopper, that applications will fail to run. Already in the late 1940's computations used subroutine libraries [8]. The use of subroutine libraries has not diminished in the past 70 years, quite to the contrary. An incredible amount of application software keeping the Internet and the world running has been written

against the POSIX-y interfaces offered by a selection of operating systems. No matter how much you do not need the obsolete features provided by the square peg operating system, you do want to keep your applications working.

From-scratch implementations of the services provided by operating systems are far from trivial undertakings. Just implementing the 20-or-so flags for the `open()` call in a real-world-bug-compatible way is far from trivial. Assuming you want to run an existing libc/application stack, you have to keep in mind that you still have roughly 199 system calls to go after `open()`. After you are done with the system calls, you then have to implement the actual components that the system calls act as an interface to: networking, file systems, device drivers, and various other driver stacks.

After the completing the above steps for a from-scratch implementation, the most time-consuming part remains: *testing your implementation in the real world and fixing it to work there*. This step is also the most difficult one, since no amount of conformance to formal specification or testing in a laboratory is a substitute for being "bug-compatible" with the real world.

So in essence, we are fascinated by square-shaped pegs because our applications rest on the support provided by those pegs. That is why we are stuck in a rut and few remember to look at the map.

1.1.3 What We Really Need

We want applications to run. We need the operating system to adapt to the scenario the application is deployed in, not for the application to adapt to a 1950's understanding of computing and hardware cost.

www.dbooks.org

Let us consider embedded systems. Your system consists of one trust-domain on one piece of hardware. There, you simply need at set of subroutines (drivers) to enable your application to run. You do not need any code which allows the single-user, single-application system to act like a timesharing system with multiple users. However, for example the implementation of the TCP/IP driver can, assuming you do not want to scale to kilobyte-sized system or to the bleeding edge of performance, be the same as one for a multiuser system. After all, the TCP/IP protocols are standard, and therefore the protocol translation the driver needs to perform is also standard.

Let us consider the cloud and especially microservices running on the cloud. We can indeed run the services on top of a timesharing operating system, A paravirtualized timesharing OS takes time to bootstrap [26] and consumes resources even for the features which are not used by the microservice. OS virtualization via containers [27] provides better performance and resource consumption than paravirtualization [53, 57], but at the cost of putting millions of lines of code into the trusted computing base.

Using timesharing systems en masse will allow applications to run in both cases, but not adapting to the scenario comes with a price. In effect, tradeoffs are made either for performance or security.

1.2 The Anykernel and Rump Kernels

This work is about how to move from the world of timesharing systems to the world of the future in a fashion in which applications continue to function. The two key terms are *anykernel* and *rump kernel*, both of which we will introduce and describe shortly.

Applications need subroutines to work, and those subroutines are provided by operating systems. We call those subroutines drivers, and state that not only does a typical application require a large set of drivers, but also that those drivers are also non-trivial to write and maintain. While operating systems built around a timesharing model are rich in drivers due to having a lot of history behind them, they are not sufficient for the use cases required by the modern world. We need to start treating drivers as library-like components instead of requiring a separate implementation for each operating system. The library-approach will allow to build the software stack to suit the scenario, instead of having to build the scenario to suit the available operating systems.

The term *anykernel* was coined in response to the ever-increasing number of operating system models: monolithic kernel, microkernel, exokernel, multikernel, unikernel, etc. As the saying goes, creating something new is is 5% inspiration and 95% perspiration. While the inspiration required to come up with a new model should not be undervalued, after that 95% of the work for reaching a usable software stack remains. That 95% consists largely of the drivers. For example, even the most trivial cloud operating system requires a TCP/IP driver, and creating one from scratch or even porting one is far from trivial. The anykernel is a term describing a kernel-type codebase from which drivers, the 95%, can be extracted and integrated to *any* operating system model — or at least near any — without porting and maintenance work.

A *rump kernel*, as the name implies, is a timesharing style kernel from which portions have been removed. What remains are drivers and the basic support routines required for the drivers to function – synchronization, memory allocators, and so forth. What is gone are policies of thread scheduling, virtual memory, application processes, and so forth. Rump kernels have a well-defined (and small!) portability layer, so they are straightforward to integrate into various environments.

www.dbooks.org

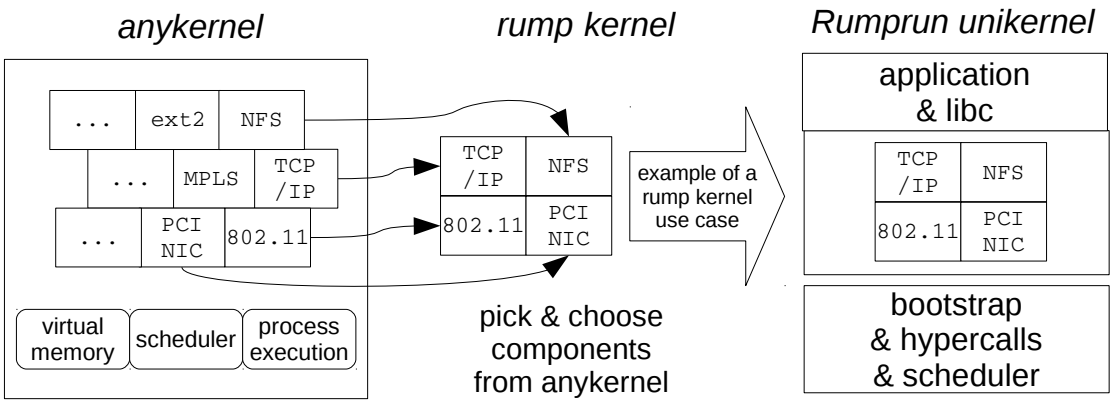


Figure 1.1: **Relationship between key concepts**: The anykernel allows driver components to be lifted out of the original source tree and rump kernels to be formed out of those drivers. Rump kernels can be used to build products and platforms; one example of a use case is illustrated.

Figure 1.1 illustrates how a timesharing system, anykernel and rump kernel are related. The figure indeed illustrates only one example, and by extension, only one example platform for hosting rump kernels.

Throughout most of the technical discussion in this book we will consider a userspace program as the platform for hosting a rump kernel. There are two reasons why it is so. First, the original motivation for rump kernels back in 2007 was developing, debugging and testing kernel drivers. What better place to do it than in userspace? Second, userspace is in itself a "hosted" platform, and we do not have full control of for example the symbol namespace or the scheduler. Therefore, if rump kernels can work in userspace, they can also easily work on platforms which are custom-built to host rump kernels.

The implementation we discuss is available in NetBSD. It is crucial to differentiate between the implementation being *in* NetBSD, and it being available as patches for NetBSD. The idea of the anykernel is that it is an inherent property of a code base, so as to keep things maintained. What, in fact, keeps the implementation working

is NetBSD's internal use of rump kernels for testing kernel drivers. This testing also allows NetBSD to provide better quality drivers, so there is clear synergy. However, we will not focus on the testing aspects in this book; if curious, see the first edition for more discussion on development and testing.

1.3 Book Outline

The remainder of the book is as follows. Chapter 2 defines the concept of the anykernel and rump kernels and Chapter 3 discusses the implementation and provides microbenchmarks as supporting evidence for implementation decisions. Essentially, the two previous chapters are a two-pass flight over the core subject. The intent is to give the reader a soft landing by first introducing the new concept in abstract terms, and then doing the same in terms of the implementation. That way, we can include discussion of worthwhile implementation details without confusing the high-level picture. If something is not clear from either chapter alone, the recommendation is to study the relevant text from the other one. If you read the first edition of this book, you may choose to only lightly skim these two chapters; the main ideas are the same as in the first edition.

Chapter 4 gives an overview of what we have built on top of rump kernels. A brief history of the project is presented in Chapter 5. The history chapter can be read first, last, or anywhere in between, or not at all. Finally, Chapter 6 provides concluding remarks.

What this book is not

This book is not a user manual. You will not learn how to use rump kernels in day-to-day operations from this book. However, you will gain a deep understanding

www.dbooks.org

of rump kernels which, when coupled with the user documentation, will give you superior knowledge of how to use rump kernels. Most of the user documentation is available as a wiki at `http://wiki.rumpkernel.org/`.

1.4 Further Material

In general, further material is reachable from the rump kernel project website at `http://rumpkernel.org/`.

1.4.1 Source Code

The NetBSD source files and their respective development histories are available for study from repository provided by the NetBSD project, e.g. via the web interface at `cvsweb.NetBSD.org`. These files are most relevant for the discussion in Chapter 2 and Chapter 3.

The easiest way to fetch the latest NetBSD source code in bulk is to run the following commands (see Section 4.1.4 for further information):

```
git clone http://repo.rumpkernel.org/src-netbsd
cd src-netbsd
git checkout all-src
```

Additionally, there is infrastructure to support building rump kernels for various platforms hosted at `http://repo.rumpkernel.org/`. The discussion in Chapter 4 is mostly centered around source code available from that location.

Code examples

This book includes code examples from the NetBSD source tree. All such examples are copyright of their respective owners and are not public domain. If pertinent, please check the full source for further information about the licensing and copyright of each such example.

1.4.2 Manual Pages

Various manual pages are cited in the document. They are available as part of the NetBSD distribution, or via the web interface at `http://man.NetBSD.org/`.

www.dbooks.org

2 Concepts: Anykernel and Rump Kernels

As a primer for the technical discussion in this book, we consider the elements that make up a modern Unix-style operating system. Commonly, the operating system is cleft in twain with the *kernel* providing basic support, and *userspace* being where applications run. Since this chapter is about the any*kernel* and rump *kernels*, we limit the following discussion to the kernel.

The *CPU specific code* is on the bottom layer of the operating system. This code takes care of low level bootstrap and provides an abstract interface to the hardware. In most, if not all, modern general purpose operating systems the CPU architecture is abstracted away from the bulk of the kernel and only the lowest layers have knowledge of it. To put the previous statement into terms which are used in our later discussions, the interfaces provided by the CPU specific code are the "hypercall" interfaces that the OS runs on. In the NetBSD kernel these functions are usually prefixed with "`cpu`".

The *virtual memory subsystem* manages the virtual address space of the kernel and application processes. Virtual memory management includes defining what happens when a memory address is accessed. Examples include normal read/write access to the memory, flagging a segmentation violation, or a file being read from the file system.

The *process execution subsystem* understands the formats that executable binaries use and knows how to create a new process when an executable is run.

The *scheduling code* includes a method and policy to define what code a CPU is executing. Scheduling can be cooperative or preemptive. Cooperative scheduling means that the currently running thread decides when to yield — sometimes this

www.dbooks.org

decision is implicit, e.g. waiting for I/O to complete. Preemptive scheduling means that also the scheduler can decide to unschedule the current thread and schedule a new one, typically because the current thread exceeded its allotted share of CPU time. When a thread is switched, the scheduler calls the CPU specific code to save the machine context of the current thread and load the context of the new thread. NetBSD uses preemptive scheduling both in userspace and in the kernel.

Atomic operations enable modifying memory atomically and avoid race conditions in for example a read-modify-write cycle. For uniprocessor architectures, kernel atomic operations are a matter of disabling interrupts and preemption for the duration of the operation. Multiprocessor architectures provide machine instructions for atomic operations. The operating system's role with atomic operations is mapping function interfaces to the way atomic operations are implemented on that particular machine architecture.

Synchronization routines such as mutexes and condition variables build upon atomic operations and interface with the scheduler. For example, if locking a mutex is attempted, the condition for it being free is atomically tested and set. If a sleep mutex was already locked, the currently executing thread interfaces with the scheduling code to arrange for itself to be put to sleep until the mutex is released.

Various *support interfaces* such CPU cross-call, time-related routines, kernel linkers, etc. provide a basis on which to build drivers.

Resource management includes general purpose memory allocation, a pool and slab [7] allocator, file descriptors, PID namespace, vmem/extent resource allocators etc. Notably, in addition to generic resources such as memory, there are more specific resources to manage. Examples of more specific resources include vnodes [30] for file systems and mbufs [58] for the TCP/IP stack.

Drivers deal with translating various protocols such as file system images, hardware devices, network packets, etc. Drivers are what we are ultimately interested in using in rump kernels, but to make them available we must deal with everything they depend on. We will touch this subject more in the next section.

2.1 Driving Drivers

To run drivers without having to run the entire timesharing OS kernel, in essence we have to provide semantically equivalent implementations of the support routines that the drivers use. The straightforward way indeed is to run the entire kernel, but it is not the optimal approach, as we argued in the introduction. The key is to figure out what to reuse verbatim and what needs rethinking.

2.1.1 Relegation and Reuse

There are essentially two problems to solve. One is coming up with an architecture which allows rump kernels to maximally integrate with the underlying platform. The second one is figuring out how to satisfy the closure of the set of support routines used by desirable drivers. Those two problems are in fact related. We will clarify in the following.

The key to drivers being able to adapt to situations is to allow them to use the features of the underlying world directly. For example, drivers need a memory address space to execute in; we use the underlying one instead of simulating a second one on top of it. Likewise, we directly use the threading and scheduling facilities in our rump kernel instead of having the scheduling a virtual kernel with its own layer of scheduling. Relegating support functionality to the host avoids adding a layer of indirection and overhead.

www.dbooks.org

Rump kernels are never full kernels which can independently run directly on bare metal, and always need lower layer support from the host. This layer can be a spartan, specially-constructed one consisting of some thousands of lines of code, or an extremely complex such as a current generation general purpose operating system.

Drivers in a rump kernel remain unmodified over the original ones. A large part of the support routines remain unmodified as well. Only in places where support is relegated to the host do we require specifically written glue code. As was indicated already in the introductory chapter, we use the term *anykernel* to describe a kernel code base with the property of being able use unmodified drivers and the relevant support routines in rump kernels.

It should be noted that unlike for example the terms *microkernel* or *unikernel*, the term *anykernel* does not convey information about how the drivers are organized at runtime, but rather that it is possible to organize them in a number of ways.

We examine the implementation details of an anykernel more closely in Chapter 3 where we turn the NetBSD kernel into an anykernel.

2.1.2 Base, Orthogonal Factions, Drivers

A monolithic kernel, as the name implies, is one single entity. The runtime footprint of a monolithic kernel contains support functionality for all subsystems, such as sockets for networking, vnodes for file systems and device autoconfiguration for drivers. All of these facilities cost resources, especially memory, even if they are not used. They may also impose dependencies on the underlying platform, e.g. MMU for some aspects of virtual memory support.

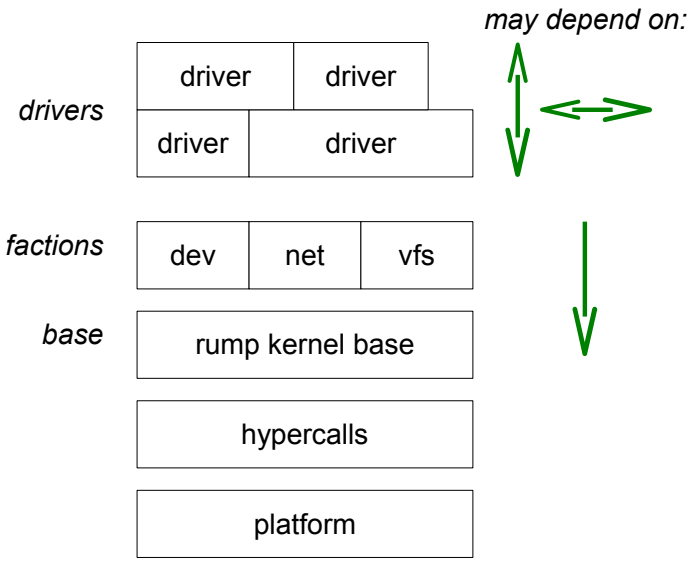


Figure 2.1: **Rump kernel hierarchy.** The desired drivers dictate the required components. The factions are orthogonal and depend only on the rump kernel base. The rump kernel base depends purely on the hypercall layer.

We have divided a rump kernel, and therefore the underlying NetBSD kernel codebase, into three layers which are illustrated in Figure 2.1: the base, factions and drivers. The base contains basic support such as memory allocation and locking. The *dev*, *net* and *vfs* factions, which denote devices, networking and [virtual] file systems, respectively, provide subsystem level support. To minimize runtime resource consumption, we require that factions are orthogonal. By orthogonal we mean that the code in one faction must be able to operate irrespective if any other faction is present in the rump kernel configuration or not. Also, the base may not depend on any faction, as that would mean the inclusion of a faction in a rump kernel is mandatory instead of optional.

We use the term *component* to describe a functional unit for a rump kernel. For example, a file system driver is a component. A rump kernel is constructed by linking together the desired set of components, either at compile-time or at run-time. A

www.dbooks.org

loose similarity exists between kernel modules and the rump kernel approach: code is compiled once per target architecture, and a linker is used to determine runtime features. For a given driver to function properly, the rump kernel must be linked with the right set of dependencies. For example, the NFS component requires both the file system and networking factions, but in contrast the tmpfs component requires only the file system faction.

User interfaces are used by applications to request services from rump kernels. Any dependencies induced by user interfaces are optional, as we will illustrate next. Consider Unix-style device driver access. Access is most commonly done through file system nodes in `/dev`, with the relevant user interfaces being *open* and *read/write* (some exceptions to the file system rule exist, such as Bluetooth and Ethernet interfaces which are accessed via sockets on NetBSD). To access a `/dev` file system node in a rump kernel, file systems must be supported. Despite file system access being the standard way to access a device, it is possible to architect an application where the device interfaces are called directly without going through file system code. Doing so means skipping the permission checks offered by file systems, calling private kernel interfaces and generally having to write more fragile code. Therefore, it is not recommended as the default approach, but if need be due to resource limitations, it is a possibility. For example, let us assume we have a rump kernel running a TCP/IP stack and we wish to use the BSD Packet Filter (BPF) [34]. Access through `/dev` is presented in Figure 2.2, while direct BPF access which does not use file system user interfaces is presented in Figure 2.3. You will notice the first example is similar to a regular application, while the latter is more complex. We will continue to refer to these examples in this chapter when we go over other concepts related to rump kernels.

The faction divisions allow cutting down several hundred kilobytes of memory overhead and milliseconds in startup time per instance. While the saving per instance is not dramatic, the overall savings are sizeable in scenarios such as IoT, network

testing [24], or cloud services, which demand thousands of instances. For example, a rump kernel TCP/IP stack without file system support is 40% smaller (400kB) than one which contains file system support.

2.1.3 Hosting

To function properly, a rump kernel must access certain underlying resources such as memory and the scheduler. These resources are accessed through the *rumpuser* hypercall interface. We will analyze and describe this interface in detail in Section 3.2.3. We call the underlying platform-specific software layer the *host*; the hypercalls are implemented on top of the host.

Notably, as we already hinted earlier, the platform requirements for a rump kernel are extremely minimal, and a rump kernel can run virtually everywhere. For example, there is no need to run the rump kernel in privileged hardware mode. Ultimately, the host has full control and fine-grained control of what a rump kernel has access to.

2.2 Rump Kernel Clients

We define a rump kernel client to be an entity which requests services from a rump kernel. Examples of rump kernel clients are userspace applications which access the network through a TCP/IP stack provided by a rump kernel, userspace applications which read files via a file system driver provided by a rump kernel, or simply any application running on the Rumprun unikernel (Section 4.2). Likewise, a test program that is used to test kernel code by means of running it in a rump kernel is a rump kernel client.

www.dbooks.org

```
int
main(int argc, char *argv[])
{
        struct ifreq ifr;
        int fd;

        /* bootstrap rump kernel */
        rump_init();

        /* open bpf device, fd is in implicit process */
        if ((fd = rump_sys_open(_PATH_BPF, O_RDWR, 0)) == -1)
                err(1, "bpf open");

        /* create virt0 in the rump kernel the easy way and set bpf to use it */
        rump_pub_virtif_create(0);
        strlcpy(ifr.ifr_name, "virt0", sizeof(ifr.ifr_name));
        if (rump_sys_ioctl(fd, BIOCSETIF, &ifr) == -1)
                err(1, "set if");

        /* rest of the application */
        [....]
}
```

Figure 2.2: **BPF access via the file system.** This figure demonstrates the system call style programming interface of a rump kernel.

```
int rumpns_bpfopen(dev_t, int, int, struct lwp *);

int
main(int argc, char *argv[])
{
        struct ifreq ifr;
        struct lwp *mylwp;
        int fd, error;

        /* bootstrap rump kernel */
        rump_init();

        /* create an explicit rump kernel process context */
        rump_pub_lwproc_rfork(RUMP_RFCFDG);
        mylwp = rump_pub_lwproc_curlwp();

        /* schedule rump kernel CPU */
        rump_schedule();

        /* open bpf device */
        error = rumpns_bpfopen(0, FREAD|FWRITE, 0, mylwp);
        if (mylwp->l_dupfd < 0) {
                rump_unschedule();
                errx(1, "open failed");
        }

        /* need to jump through a hoop due to bpf being a "cloning" device */
        error = rumpns_fd_dupopen(mylwp->l_dupfd, &fd, 0, error);
        rump_unschedule();
        if (error)
                errx(1, "dup failed");

        /* create virt0 in the rump kernel the easy way and set bpf to use it */
        rump_pub_virtif_create(0);
        strlcpy(ifr.ifr_name, "virt0", sizeof(ifr.ifr_name));
        if (rump_sys_ioctl(fd, BIOCSETIF, &ifr) == -1)
                err(1, "set if");

        /* rest of the application */
        [....]
}
```

Figure 2.3: **BPF access without a file system.** This figure demonstrates the ability to directly call arbitrary kernel routines from a user program. For comparison, it implements the same functionality as Figure 2.2. This ability is most useful for writing kernel unit tests when the calls to the unit under test cannot be directly invoked by using the standard system call interfaces.

www.dbooks.org

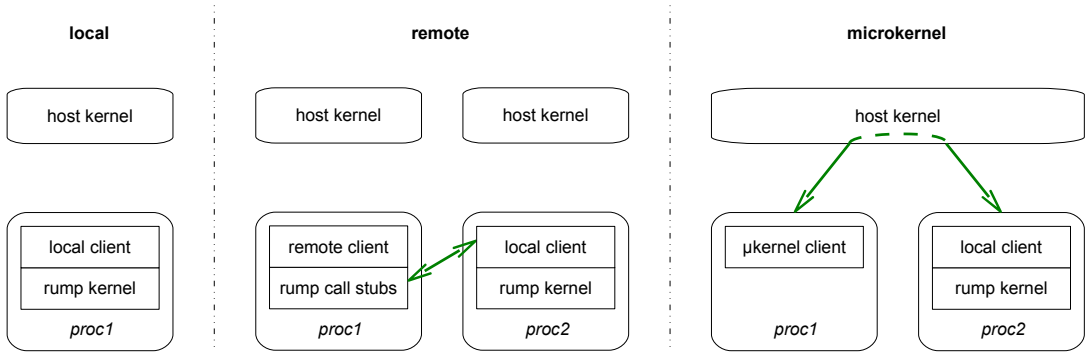


Figure 2.4: **Client types illustrated.** For local clients the client and rump kernel reside in a single process, while remote and microkernel clients reside in separate processes and therefore do not have direct memory access into the rump kernel.

The relationship between a rump kernel and a rump kernel client is an almost direct analogy to an application process executing on an operating system and requesting services from the host kernel.

There are several possible relationship types the client and rump kernel can have. Each of them have different implications on the client and kernel. The possibilities are: *local*, *remote* and *microkernel*. The configurations are also depicted in Figure 2.4. The implications of each are available in summarized form in Table 2.1. Next, we will discuss the configurations and explain the table.

- **Local** clients exist in the same application process as the rump kernel itself. They have full access to the rump kernel's address space, and make requests via function calls directly into the rump kernel. Typically requests are done via established interfaces such as the rump kernel syscall interface, but there is nothing preventing the client from jumping to any routine inside the rump kernel.

  The benefits of local clients include speed and compactness. Speed is due to a rump kernel request being essentially a function call. A null rump kernel

Type	Request Policy	Access	Available Interface
local	client	full	all
remote	client	limited	system call
microkernel	host kernel	limited	depends on service

Table 2.1: **Comparison of client types.** Local clients get full access to a rump kernel, but require explicit calls in the program code. Remote clients have standard system call access with security control and can use unmodified binaries. In microkernel mode, the rump kernel is run as a microkernel style system server with requests routed by the host kernel.

system call is twice as fast as a native system call. Compactness results from the fact that there is only a single program and can make managing the whole easier. The drawback is that the single program must configure the kernel to a suitable state before the application can act. Examples of configuration tasks include adding routing tables (the `route` utility) and mounting file systems (the `mount` utility). Since existing configuration tools are built around the concept of executing different configuration steps as multiple invocations of the tool, adaptation of the configuration code may not always be simple.

Local clients do not have meaningful semantics for a host `fork()` call. This lack of semantics is because the rump kernel state would be duplicated and could result in for example two kernels accessing the same file system or having the same IP address.

A typical example of a local client is an application which uses the rump kernel as a programming library e.g. to access a file system.

- **Remote** clients use a rump kernel which resides elsewhere, either in a different address space on the local host or on a remote one. The request routing policy is up to the client. The policy locus is an implementation decision, not a design decision, and alternative implementations can be considered [20] if it is important to have the request routing policy outside of the client.

www.dbooks.org

Since the client and kernel are separated, kernel side access control is fully enforced — if the client and rump kernel are on the same host, we assume that the host enforces separation between the respective processes. This separation means that a remote client will not be able to access resources except where the rump kernel lets it, and neither will it be able to dictate the thread and process context in which requests are executed. The client not being able to access arbitrary kernel resources in turn means that real security models are possible, and that different clients may have varying levels of privileges.

By default, we provide support for remote clients which communicate with a rump kernel using host local domain sockets or TCP sockets. Using sockets is not the only option on general purpose operating systems, and for example the `ptrace()` facility can also be used to implement remote clients [16, 20]. Also, we know that the protocol can be implemented over various media in non-POSIX environments, e.g. over a hardware bus.

Remote clients are not as performant as local clients due to IPC overhead. However, since multiple remote clients can run against a single rump kernel, they lead to more straightforward use of existing code and even that of unmodified binaries. Such binaries can be useful to configure and inspect a rump kernel (e.g. `ifconfig`). We discuss these binaries and their advantages further in Section 3.12 and Section 4.3.

Remote clients, unlike local clients, have meaningful semantics for `fork()` since both the host kernel context and rump kernel contexts can be correctly preserved: the host `fork()` duplicates only the client and not the rump kernel. Of course, this statement applies only to hosts which support a `fork()` call.

- **Microkernel** client requests are routed by the host kernel to a separate server which handles the requests using a driver in a rump kernel. While microkernel clients can be seen to be remote clients, the key difference to

remote clients is that the request routing policy is in the host kernel instead of in the client. Furthermore, the interface used to access the rump kernel is below the system call layer. We implemented microkernel callbacks for file systems (puffs [28]) and *character/block* device drivers (pud [43]). They use the NetBSD kernel VFS/vnode and *cdev/bdev* interfaces to access the rump kernel, respectively.

It needs to be noted that rump kernels accepting multiple different types of clients are possible. For example, remote clients can be used to configure a rump kernel, while the application logic still remains in the local client. The ability to use multiple types of clients on a single rump kernel makes it possible to reuse existing tools for the configuration job and still reap the speed benefit of a local client.

Rump kernels used by remote or microkernel clients always include a local client as part of the process the rump kernel is hosted in. This local client is responsible for forwarding incoming requests to the rump kernel, and sending the results back after the request has been processed.

2.3 Threads and Schedulers

Next, we will discuss the theory and concepts related to processes, threads, CPUs, scheduling and interrupts in a rump kernel. An example scenario is presented after the theory in Section 2.3.4. This subject is revisited in Section 3.3 where we discuss it from a more concrete perspective along with the implementation.

As stated earlier, a rump kernel uses the host's thread and scheduling facilities. To understand why we still need to discuss this topic, let us first consider what a thread represents to an operating system. First, a thread represents machine execution

www.dbooks.org

context, such as the program counter, other registers and the virtual memory address space. We call this machine context the *hard context*. It determines how machine instructions will be executed when a thread is running on a CPU and what their effects will be. The hard context is determined by the platform that the thread runs on. Second, a thread represents all auxiliary data required by the operating system. We call this auxiliary data the *soft context*. It comprises for example of information determining which process a thread belongs to, and e.g. therefore what credentials and file descriptors it has. The soft context is determined by the operating system.

To further illustrate, we go over a simplified version of what happens on NetBSD when an application process creates a thread:

1. The application calls `pthread_create()` and passes in the necessary parameters, including the address of the new thread's start routine.

2. The pthread library does the necessary initialization, including stack allocation. It creates a hard context by calling `_lwp_makecontext()` and passing the start routine's address as an argument. The pthread library then invokes the `_lwp_create()` system call.

3. The host kernel creates the kernel soft context for the new thread and the thread is put into the run queue.

4. The newly created thread will be scheduled and begin execution at some point in the future.

A rump kernel uses host threads for the hard context. Local client threads which call a rump kernel are created as described above. Since host thread creation does not involve the rump kernel, a host thread does not get an associated rump kernel thread soft context upon creation.

Nonetheless, a unique rump kernel soft context must exist for each thread executing within the rump kernel because the code we wish to run relies on it. For example, code dealing with file descriptors accesses the relevant data structure by dereferencing `curlwp->l_fd` [1]. The soft context determines the value of `curlwp`.

We must solve the lack of a rump kernel soft context resulting from the use of host threads. Whenever a host thread makes a function call into the rump kernel, an entry point wrapper must be called. Conversely, when the rump kernel routine returns to the client, an exit point wrapper is called. These calls are done automatically for official interfaces, and must be done manually in other cases — compare Figure 2.2 and Figure 2.3 and see that the latter includes calls to `rump_schedule()` and `rump_unschedule()`. The wrappers check the host's thread local storage (TLS) to see if there is a rump kernel soft context associated with the host thread. The soft context may either be set or not set. We discuss both cases in the following paragraphs.

1. **implicit threads**: the soft context is not set in TLS. A soft context will be created dynamically and is called an *implicit thread*. Conversely, the implicit thread will be released at the exit point. Implicit threads are always attached to the same rump kernel process context, so callers performing multiple calls, e.g. opening a file and reading from the resulting file descriptor, will see expected results. The rump kernel thread context will be different as the previous one no longer exists when the next call is made. A different context does not matter, as the kernel thread context is not exposed to userspace through any portable interfaces — that would not make sense for systems which implement a threading model where userspace threads are multiplexed on top of kernel provided threads [2].

[1] `curlwp` is not variable in the C language sense. It is a platform-specific macro which produces a pointer to the currently executing thread's kernel soft context. Furthermore, since file descriptors are a process concept instead of a thread concept, it would be more logical to access them via `curlwp->l_proc->p_fd`. This commonly referenced pointer is cached directly in the thread structure as an optimization to avoid indirection.

www.dbooks.org

2. **bound threads**: the soft context is set in TLS. The rump kernel soft context in the host thread's TLS can be set, changed and disbanded using interfaces further described in the manual page *rump_lwproc.3*. We call a thread with the rump kernel soft context set a *bound thread*. All calls to the rump kernel made from a host thread with a bound thread will be executed with the same rump kernel soft context.

The soft context is always set by a local client. Microkernel and remote clients are not able to directly influence their rump kernel thread and process context. Their rump kernel context is set by the local client which receives the request and makes the local call into the rump kernel.

Discussion

There are alternative approaches to implicit threads. It would be possible to require all local host threads to register with the rump kernel before making calls. The registration would create essentially a bound thread. There are two reasons why this approach was not chosen. First, it increases the inconvenience factor for casual users, e.g. in kernel testing use cases, as now a separate call per host thread is needed. Second, some mechanism like implicit threads must be implemented anyway: allocating a rump kernel thread context requires a rump kernel context for example to be able to allocate memory for the data structures. Our implicit thread implementation doubles as a bootstrap context.

Implicit contexts are created dynamically because because any preconfigured reasonable amount of contexts risks application deadlock. For example, n implicit threads can be waiting inside the rump kernel for an event which is supposed to be delivered by the $n + 1$'th implicit thread, but only n implicit threads were precreated. Creating an amount which will never be reached (e.g. 10,000) may avoid deadlock,

but is wasteful. Additionally, we assume all users aiming for high performance will use bound threads.

Notably, in some rump kernel use cases where rump kernel and host threads are always 1:1-mapped, such as with the Rumprun unikernel, all threads are established as bound threads transparently to the applications. However, the implicit thread mechanism is still used to bootstrap the contexts for those threads.

2.3.1 Kernel threads

Up until now, we have discussed the rump kernel context of threads which are created by the client, e.g. by calling `pthread_create()` on a POSIX host. In addition, *kernel threads* exist. The creation of a kernel thread is initiated by the kernel and the entry point lies within the kernel. Therefore, a kernel thread always executes within the kernel except when it makes a hypercall. Kernel threads are associated with process 0 (`struct proc0`). An example of a kernel thread is the *workqueue worker* thread, which the workqueue kernel subsystem uses to schedule and execute asynchronous work units.

On a regular system, both an application process thread and a kernel thread have their hard context created by the kernel. As we mentioned before, a rump kernel cannot create a hard context. Therefore, whenever kernel thread creation is requested, the rump kernel creates the soft context and uses a hypercall to request the hard context from the host. The entry point given to the hypercall is a bouncer routine inside the rump kernel. The bouncer first associates the kernel thread's soft context with the newly created host thread and then proceeds to call the thread's actual entry point.

www.dbooks.org

2.3.2 A CPU for a Thread

First, let us use broad terms to describe how scheduling works in regular virtualized setup. The hypervisor has an idle CPU it wants to schedule work onto and it schedules a guest system. While the guest system is running, the guest system decides which guest threads to run and when to run them using the guest system's scheduler. This means that there are two layers of schedulers involved in scheduling a guest thread.

We also point out that a guest CPU can be a purely virtual entity, e.g. the guest may support multiplexing a number of virtual CPUs on top of one host CPU. Similarly, the rump kernel may be configured to provide any number of CPUs that the guest OS supports regardless of the number of CPUs present on the host. The default for a rump kernel is to provide the same number of virtual CPUs as the number of physical CPUs on the host. Then, a rump kernel can fully utilize all the host's CPUs, but will not waste resources on virtual CPUs where the host cannot schedule threads for them in parallel.

As a second primer for the coming discussion, we will review CPU-local algorithms. CPU-local algorithms are used avoid slow cross-CPU locking and hardware cache invalidation. Consider a pool-style resource allocator (e.g. memory): accessing a global pool is avoided as far as possible because of the aforementioned reasons of locking and cache. Instead, a CPU-local allocation cache for the pools is kept. Since the local cache is tied to the CPU, and since there can be only one thread executing on one CPU at a time, there is no need for locking other than disabling thread preemption in the kernel while the local cache is being accessed. Figure 2.5 gives an illustrative example.

The host thread doubles as the guest thread in a rump kernel and the host schedules guest threads. The guest CPU is left out of the relationship. The one-to-one

```
void *
pool_cache_get_paddr(pool_cache_t pc)
{
    pool_cache_cpu_t *cc;

    cc = pc->pc_cpus[curcpu()->ci_index];
    pcg = cc->cc_current;
    if (__predict_true(pcg->pcg_avail > 0)) {
            /* fastpath */
            object = pcg->pcg_objects[--pcg->pcg_avail].pcgo_va;
            return object;
    } else {
            return pool_cache_get_slow();
    }
}
```

Figure 2.5: **Use of `curcpu()` in the pool allocator** simplified as pseudocode from `sys/kern/subr_pool.c`. An array of CPU-local caches is indexed by the current CPU's number to obtain a pointer to the CPU-local data structure. Lockless allocation from this cache is attempted before reaching into the global pool.

relationship between the guest CPU and the guest thread must exist because CPU-local algorithms rely on that invariant. If we remove the restriction of each rump kernel CPU running at most one thread at a time, code written against CPU-local algorithms will cause data structure corruption. Therefore, it is necessary to uphold the invariant that a CPU has at most one thread executing on it at a time.

Since selection of the guest thread is handled by the host, we select the guest CPU instead. The rump kernel virtual CPU is assigned for the thread that was selected by the host, or more precisely that thread's rump kernel soft context. Simplified, scheduling in a rump kernel can be considered picking a CPU data structure off of a freelist when a thread enters the rump kernel and returning the CPU to the freelist once a thread exits the rump kernel. A performant implementation is more delicate due to multiprocessor efficiency concerns. One is discussed in more detail along with the rest of the implementation in Section 3.3.1.

www.dbooks.org

Scheduling a CPU and releasing it are handled at the rump kernel entrypoint and exitpoint, respectively. The BPF example with VFS (Figure 2.2) relies on rump kernel interfaces handling scheduling automatically for the clients. The BPF example which calls kernel interfaces directly (Figure 2.3) schedules a CPU before it calls a routine inside the rump kernel.

2.3.3 Interrupts and Preemption

An interrupt is an asynchronously occurring event which preempts the current thread and proceeds to execute a compact handler for the event before returning control back to the original thread. The interrupt mechanism allows the OS to quickly acknowledge especially hardware events and schedule the required actions for a suitable time (which may be immediately). Taking an interrupt is tied to the concept of being able to temporarily replace the currently executing thread with the interrupt handler. Kernel thread preemption is a related concept in that code currently executing in the kernel can be removed from the CPU and a higher priority thread selected instead.

The rump kernel uses a cooperative scheduling model where the currently executing thread runs to completion. There is no virtual CPU preemption, neither by interrupts nor by the scheduler. A thread holds on to the rump kernel virtual CPU until it either makes a blocking hypercall or returns from the request handler. A host thread executing inside the rump kernel may be preempted by the host. Preemption will leave the virtual CPU busy until the host reschedules the preempted thread and the thread runs to completion in the rump kernel.

What would be delivered by a preempting interrupt in the monolithic kernel is always delivered via a schedulable thread in a rump kernel. In the event that later use cases present a strong desire for fast interrupt delivery and preemption, the

author's suggestion is to create dedicated virtual rump CPUs for interrupts and real-time threads and map them to high-priority host threads. Doing so avoids interaction with the host threads via signal handlers (or similar mechanisms on other non-POSIX host architectures). It is also in compliance with the paradigm that the host handles all scheduling in a rump kernel.

2.3.4 An Example

We present a clarifying example. Let us assume two host threads, A and B, which both act as local clients. The host schedules thread A first. It makes a call into the rump kernel requesting a bound thread. First, the soft context for an implicit thread is created and a CPU is scheduled. The implicit thread soft context is used to create the soft context of the bound thread. The bound thread soft context is assigned to thread A and the call returns after free'ing the implicit thread and releasing the CPU. Now, thread A calls the rump kernel to access a driver. Since it has a bound thread context, only CPU scheduling is done. Thread A is running in the rump kernel and it locks mutex M. Now, the host scheduler decides to schedule thread B on the host CPU instead. There are two possible scenarios:

1. The rump kernel is a uniprocessor kernel and thread B will be blocked. This is because thread A is still scheduled onto the only rump kernel CPU. Since there is no preemption for the rump kernel context, B will be blocked until A runs and releases the rump kernel CPU. Notably, it makes no difference if thread B is an interrupt thread or not — the CPU will not be available until thread A releases it.

2. The rump kernel is a multiprocessor kernel and there is a chance that other rump kernel CPUs may be available for thread B to be scheduled on. In this case B can run.

www.dbooks.org

We assume that B can run immediately. Thread B uses implicit threads, and therefore upon entering the rump kernel an implicit thread soft context gets created and assigned to thread B, along with a rump kernel CPU.

After having received a rump kernel CPU and thread context, thread B wants to lock mutex M. M is held, and thread B will have to block and await M's release. Thread B will release the rump kernel CPU and sleep until A unlocks the mutex. After the mutex is unlocked, the host marks thread B as runnable and after B wakes up, it will attempt to schedule a rump kernel CPU and after that attempt to lock mutex M and continue execution. When B is done with the rump kernel call, it will return back to the application. Before doing so, the CPU will be released and the implicit thread context will be free'd.

Note that for thread A and thread B to run in *parallel*, both the host and the rump kernel must have multiprocessor capability. If the host is uniprocessor but the rump kernel is configured with multiple virtual CPUs, the threads can execute inside the rump kernel *concurrently*. In case the rump kernel is configured with only one CPU, the threads will execute within the rump kernel *sequentially* irrespective of if the host has one or more CPUs available for the rump kernel.

2.4 Virtual Memory

Virtual memory address space management in a rump kernel (if any!) is relegated to the host, because requiring support in a rump kernel would impose restrictions on the platform and host. For example, emulating page faults and memory protection in a usermode OS exhibits over tenfold performance penalty and can be significant in other, though not all, hypervisors [3]. Virtual memory support was not seen worth the increased implementation complexity and potentially reduced performance.

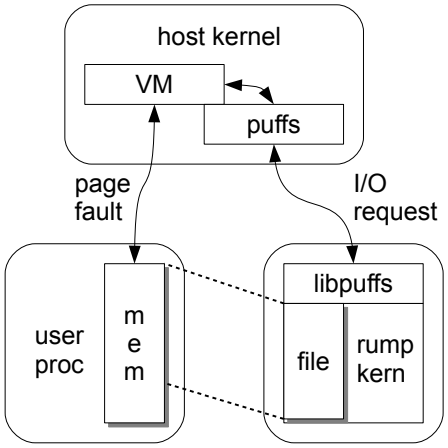


Figure 2.6: **Providing memory mapping support on top of a rump kernel**. The file is mapped into the client's address space by the host kernel. When non-resident pages in the mapped range are accessed by the client, a page fault is generated and the rump kernel is invoked via the host kernel's file system code to supply the desired data.

The implication of a rump kernel not implementing full memory protection is that it does not support accessing resources via page faults. There is no support in a rump kernel for memory mapping a file to a client. Supporting page faults *inside* a rump kernel would not work for remote clients anyway, since the page faults need to be trapped on the client machine.

However, it is possible to provide memory mapping support *on top* of rump kernels. In fact, when running file systems as microkernel servers, the puffs [28] userspace file system framework and the host kernel provide memory mapping support for the microkernel client. The page fault is resolved in the host kernel, and the I/O request for paging in the necessary data sent to the rump kernel. After the rump kernel has satisfied the request and responded via puffs, the host kernel unblocks the process that caused the page fault (Figure 2.6). If a desirable use case is found, distributed shared memory [40] can be investigated for memory mapping support in remote clients.

www.dbooks.org

Another implication of the lack of memory protection is that a local client can freely access the memory in a rump kernel. Consider the BPF example which accesses the kernel directly (Figure 2.3). Not only does the local client call kernel routines, it also examines the contents of a kernel data structure.

2.5 Distributed Services with Remote Clients

As mentioned in our client taxonomy in Section 2.2, remote clients use services from a rump kernel hosted either on the same host in another process or on a remote host. We describe the general concept here and provide implementation details later in Section 3.12.

It is known to be possible to build a Unix system call emulation library on top of a distributed system [41]. We go further: while we provide the Unix interface to applications, we also use existing Unix kernel code at the server side.

Running a client and the rump kernel on separate hosts is possible because on a fundamental level Unix already works like a distributed system: the kernel and user processes live in different address spaces and information is explicitly moved across this boundary by the kernel. Copying data across the boundary simplifies the kernel, since data handled by the kernel can always be assumed to be resident and non-changing. Explicit copy requests in the kernel code make it possible to support remote clients by implementing only a request transport layer. System calls become RPC requests from the client to the kernel and routines which copy data between arbitrary address spaces become RPC requests from the kernel to the client.

When a remote client connects to a rump kernel, it gets assigned a rump kernel process context with appropriate credentials. After the handshake is complete, the remote client can issue service requests via the standard system call interface. First,

the client calls a local stub routine, which marshals the request. The stub then sends the request to the server and blocks the caller. After the rump kernel server has processed the request and responded, the response is decoded and the client is unblocked. When the connection between a rump kernel and a remote client is severed, the rump kernel treats the client process as terminated.

The straightforward use of existing data structures has its limitations: the system the client is hosted on must share the same ABI with the system hosting the rump kernel. Extending support for systems which are not ABI-compatible is beyond the scope of our work. However, working remote client support shows that it is possible to build distributed systems out of a Unix codebase without the need for a new design and codebase such as Plan 9 [46].

2.6 Summary

Rump kernels provide lightweight driver stacks which can run on practically any platform. To be as lightweight and portable as possible, rump kernels rely on two features: relegating support functionality to the host and an anykernel codebase where different units of the kernel (e.g. networking and file systems) are disjoint enough to be usable in configurations where all parties are not present.

Rump kernels support three types of clients: local, microkernel and remote. Each client type has its unique properties and varies for example in access rights to a rump kernel, the mechanism for making requests, and performance characteristics. Remote clients are able to access a rump kernel over the Internet and other media.

For drivers to function, a rump kernel must possess runtime context information. This information consists of the process/thread context and a unique rump kernel CPU that each thread is associated with. A rump kernel does not assume virtual

www.dbooks.org

memory, and does not provide support for page faults or memory protection. Virtual memory protection and page faults, where necessary, are always left to be performed by the host of the rump kernel client.

3 Implementation: Anykernel and Rump Kernels

The previous chapter discussed the concept of an anykernel and rump kernels. This chapter describes the code level modifications that were necessary for a production quality implementation on NetBSD. The terminology used in this chapter is mainly that of NetBSD, but the concepts apply to other similar operating systems as well.

In this chapter we reduce the number of variables in the discussion by limiting our examination to rump kernels and their clients running on a NetBSD host. See the next chapter (Chapter 4) for discussion on rump kernels and their clients running on hosts beyond NetBSD userspace.

3.1 Kernel Partitioning

As mentioned in Section 2.1.2, to maximize the lightweight nature of rump kernels, the kernel code was several logical layers: a base, three factions (dev, net and vfs) and drivers. The factions are orthogonal, meaning they do not depend on each other. Furthermore, the base does not depend on any other part of the kernel. The modifications we made to reach this goal of independence are described in this section.

As background, it is necessary to recall how the NetBSD kernel is linked. In C linkage, symbols which are unresolved at compile-time must be satisfied at binary link-time. For example, if a routine in `file1.c` wants to call `myfunc()` and `myfunc()` is not present in any of the object files or libraries being linked into a binary, the linker flags an error. A monolithic kernel works in a similar fashion: all symbols must be resolved when the kernel is linked. For example, if an object file with an unresolved symbol to the kernel's pathname lookup routine `namei()` is included,

www.dbooks.org

then either the symbol `namei` must be provided by another object file being linked, or the calling source module must be adjusted to avoid the call. Both approaches are useful for us and the choice depends on the context.

We identified three obstacles for having a partitioned kernel:

1. **Compile-time definitions** (`#ifdef`) indicating which features are present in the kernel. Compile-time definitions are fine within a component, but do not work between components if linkage dependencies are created (for example a cross-component call which is conditionally included in the compilation).
2. **Direct references** between components where we do not allow them. An example is a reference from the base to a faction.
3. **Multiclass source modules** contain code which logically belongs in several components. For example, if the same file contains routines related to both file systems and networking, it belongs in this problem category.

Since our goal is to change the original monolithic kernel and its characteristics as little as possible, we wanted to avoid heavy approaches in addressing the above problems. These approaches include but are not limited to converting interfaces to be called only via pointer indirection. Instead, we observed that indirect interfaces were already used on most boundaries (e.g. `struct fileops`, `struct protosw`, etc.) and we could concentrate on the exceptions. Code was divided into functionality groups using source modules as boundaries.

The two techniques we used to address problems are as follows:

1. **code moving**. This solved cases where a source module belonged to several classes. Part of the code was moved to another module. This technique had

to be used sparingly since it is very intrusive toward other developers who have outstanding changes in their local trees. However, we preferred moving over splitting a file into several portions using `#ifdef`, as the final result is clearer to anyone looking at the source tree.

In some cases code, moving had positive effects beyond rump kernels. One such example was splitting up `sys/kern/init_sysctl.c`, which had evolved to include *sysctl* handlers for many different pieces of functionality. For example, it contained the routines necessary to retrieve a process listing. Moving the process listing routines to the source file dealing with process management (`sys/kern/kern_proc.c`) not only solved problems with references to factions, but also grouped related code and made it easier to locate.

2. **function pointers**. Converting direct references to calls via function pointers removes link-time restrictions. A function pointer gets a default value at compile time. Usually this value is a stub indicating the requested feature is not present. At runtime the pointer may be adjusted to point to an actual implementation of the feature if it is present.

Previously, we also used weak symbol aliases sparingly to provide stub implementations which were overridden by the linker if the component providing the actual implementation was linked. Weak aliases were found to be problematic with dynamically linked libraries on some userspace platforms, e.g. Linux with glibc. In *lazy binding*, a function is resolved by the dynamic linker only when the function is first called, so as to avoid long program startup times due to resolving symbols which are never used at runtime. As a side-effect, lazy binding theoretically allows `dlopen()`'ing libraries which override weak aliases as long as the libraries are loaded before the overridden functions are first called. Namely, in the case of rump kernels, loading must take place before `rump_init()` is called. However, some dynamic linkers treat libraries loaded with `dlopen()` different from ones loaded when the binary is executed. For example, the aforementioned glibc dynamic linker overrides

www.dbooks.org

weak aliases with symbols from `dlopen()`'d libraries only if the environment variable `LD_DYNAMIC_WEAK` is set. With some other dynamic linkers, overriding weak symbols is not possible at all. Part of the power of rump kernels is the ability to provide a single binary which dynamically loads the necessary components at runtime depending on the configuration or command line parameters. Therefore, to ensure that rump kernels work the same on all userspace platforms, we took the extra steps necessary to remove uses of weak aliases and replace them with the above-mentioned two techniques.

To illustrate the problems and our necessary techniques, we discuss the modifications to the file `sys/kern/kern_module.c`. The source module in question provides support for loadable kernel modules (discussed further in Section 3.8.1). Originally, the file contained routines both for loading kernel modules from the file system and for keeping track of them. Having both in one module was a valid possibility before the anykernel faction model. In the anykernel model, loading modules from a file system is VFS functionality, while keeping track of the modules is base functionality.

To make the code comply with the anykernel model, we used the code moving technique to move all code related to file system access to its own source file in `kern_module_vfs.c`. Since loading from a file system must still be initiated by the kernel module management routines, we introduced a function pointer interface. By default, it is initialized to a stub:

```
int (*module_load_vfs_vec)(const char *, int, bool, module_t *,
                           prop_dictionary_t *) = (void *)eopnotsupp;
```

If VFS is present, the routine `module_load_vfs_init()` is called during VFS subsystem init after the `vfs_mountroot()` routine has successfully completed to set the value of the function pointer to `module_load_vfs()`. In addition to avoiding a

direct reference from the base to a faction in rump kernels, this pointer has another benefit: during bootstrap it protects the kernel from accidentally trying to load kernel modules from the file system before the root file system has been mounted [2].

3.1.1 Extracting and Implementing

We have two methods for providing functionality in the rump kernel: we can *extract* it out of the kernel sources, meaning we use the source file as such, or we can *implement* it, meaning that we do an implementation suitable for use in a rump kernel. We work on a source file granularity level, which means that either all of an existing source file is extracted, or the necessary routines from it (which may be all of them) are implemented. Implemented source files are placed under `sys/rump`, while extracted ones are picked up by Makefiles from other subdirectories under `sys/`.

The goal is to extract as much as possible for the features we desire, as to minimize implementation and maintenance effort and to maximize the semantic correctness of the used code. Broadly speaking, there are three cases where extraction is not possible.

1. **code that does not exist in the regular kernel**: this means drivers specific to rump kernels. Examples include anything using rump hypercalls, such as the virtual block device driver.

2. **code dealing with concepts not supported in rump kernels**. An example is the virtual memory fault handler: when it is necessary to call a routine which in a regular kernel is invoked from the fault hander, it must be done from implemented code.

[2] `sys/kern/vfs_subr.c` rev 1.401

www.dbooks.org

It should be noted, though, that not all VM code should automatically be disqualified from extraction. For instance, VM readahead code is an algorithm which does not have anything per se to do with virtual memory, and we have extracted it from `sys/uvm/uvm_readahead.c`.

3. **bypassed layers** such as scheduling. They need different handling.

In some cases a source module contained code which was desirable to be extracted, but it was not possible to use the whole source module because others parts were not suitable for extraction. Here we applied the code moving technique. As an example, we once again look at the code dealing with processes (`kern_proc.c`). The source module contained mostly process data structure management routines, e.g. the routine for mapping a process ID number (`pid_t`) to the structure describing it (`struct proc *`). We were interested in being able to extract this code. However, the same file also contained the definition of the `lwp0` variable. Since that definition included references to the scheduler ("concept not supported in a rump kernel"), we could not extract the file as such. However, after moving the definition of `lwp0` to `kern_lwp.c`, where it arguably belongs, `kern_proc.c` could be extracted.

3.1.2 Providing Components

We provide components as libraries. The kernel base library is called `librump` and the hypervisor library is called `librumpuser`. The factions are installed with the names `librumpdev`, `librumpnet` and `librumpvfs` for dev, net and vfs, respectively. The driver components are named with the pattern `librump<faction>_driver`, e.g. `librumpfs_nfs` (NFS client driver). The faction part of the name is an indication of what type of driver is in question, but it does not convey definitive information on what the driver's dependencies are. For example, consider the NFS client: while it is a file system driver, it also depends on networking.

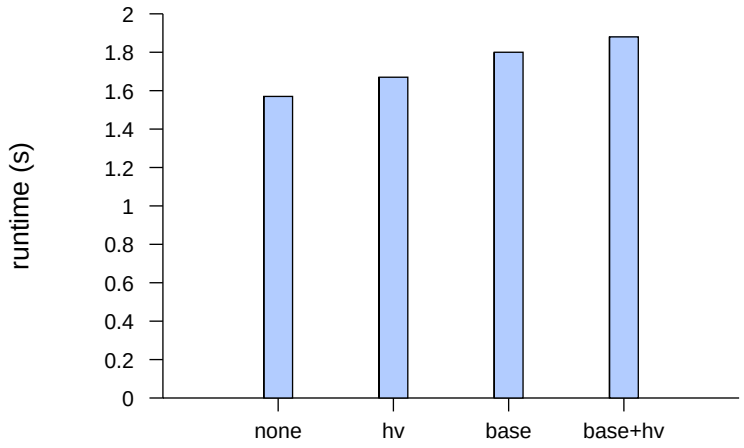


Figure 3.1: **Performance of position independent code (PIC).** A regular kernel is compiled as non-PIC code. This compilation mode is effectively the same as "none" in the graph. If the hypervisor and rump kernel base use PIC code, the execution time increases as is expected. In other words, rump kernels allow to make a decision on the tradeoff between execution speed and memory use.

Two types of libraries are available: static and dynamic. Static libraries are linked by the toolchain into the binary, while dynamic binaries are linked at runtime. Commonly, dynamic linking is used with shared libraries compiled as position independent code (PIC), so as to allow one copy of a library to be resident in memory and be mapped into an arbitrary number of programs. Rump kernels support both types of libraries, but it needs to be noted that dynamic linking depends on the host supporting that runtime feature. It also need to be noted that while shared libraries save memory in case they are needed more than once, they have inherently worse performance due to indirection [21]. Figure 3.1 illustrates that performance penalty by measuring the time it takes to create and disband 300k threads in a rump kernel. As can be deduced from the combinations, shared and static libraries can be mixed in a single rump kernel instance so as to further optimize the behavior with the memory/CPU tradeoff.

www.dbooks.org

3.2 Running the Kernel in an Hosted Environment

Software always runs on top of an entity which provides the interfaces necessary for the software to run. A typical operating system kernel runs on top of hardware and uses the hardware's "interfaces" to fulfill its needs. When running on top a hardware emulator the emulator provides the same hardware interfaces. In a paravirtualized setup the hypervisor provides the necessary interfaces. In a usermode OS setup, the application environment of the hosting OS makes up the hypervisor. In this section we discuss details related to hosting a rump kernel in any environment. We use POSIX userspace as the case for the bulk of the discussion, since that host induces the superset of issues related to hosting rump kernels.

3.2.1 C Symbol Namespaces

In the regular case, the kernel and userspace C namespaces are disjoint. Both the kernel and application can contain the same symbol name, for example `printf`, without a collision occurring. When we run the kernel in a process container, we must take care to preserve this property. Calls to printf made by the client still need to go to libc, while calls to printf made by the rump kernel need to be handled by the in-kernel implementation.

Single address space operating systems provide a solution [12], but require a different calling convention. On the other hand, C preprocessor macros were used by OSKit [18] to rename conflicting symbols and allow multiple different namespaces to be linked together. UML [16] uses a variant of the same technique and renames colliding symbols using a set of preprocessor macros in the kernel build Makefile, e.g. `-Dsigprocmask=kernel_sigprocmask`. This manual renaming approach is inadequate for a rump kernel; unlike a usermode OS kernel which is an executable application, a rump kernel is a library which is compiled, shipped, and may be linked

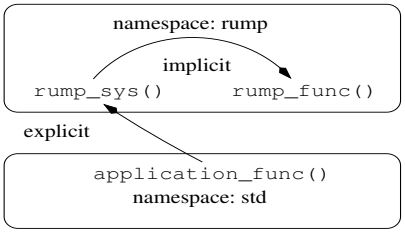


Figure 3.2: **C namespace protection**. When referencing a rump kernel symbol from outside of the rump kernel, the prefix must be explicitly included in the code. All references from inside the rump kernel implicitly contain the prefix due to bulk symbol renaming. Corollary: it is not possible to access a symbol outside the rump kernel namespace from inside the rump kernel without using a hypercall.

with any other libraries afterwards. This set of libraries is not available at compile time and therefore we cannot know which symbols will cause conflicts at link time. Therefore, the only option is to assume that any symbol may cause a conflict.

We address the issue by protecting all symbols within the rump kernel. The `objcopy` utility's rename functionality is used ensure that all symbols within the rump kernel have a prefix starting with "rump" or "RUMP". Symbol names which do not begin with "rump" or "RUMP" are renamed to contain the prefix "rumpns_". After renaming, the kernel `printf` symbol will be seen as `rumpns_printf` by the linker. Prefixes are illustrated in Figure 3.2: callers outside of a rump kernel must include the prefix explicitly, while the prefix for routines inside a rump kernel is implicit since it is automatically added by `objcopy`. Table 3.1 illustrates further by providing examples of the outcome of renaming.

However, renaming all symbols also creates a problem. Not all symbols in a kernel object file come from kernel source code. Some symbols are a property of the toolchain. An example is `_GLOBAL_OFFSET_TABLE_`, which is used by position independent code to store the offsets. Renaming toolchain-generated symbols causes failures, since the toolchain expects to find symbols where it left them.

www.dbooks.org

rump kernel object	original symbol name	symbol after renaming
yes	`rump_sys_call`	`rump_sys_call`
yes	`printf`	`rumpns_printf`
no	`rump_sys_call`	`rump_sys_call`
no	`printf`	`printf`

Table 3.1: **Symbol renaming illustrated.** Objects belonging to a rump kernel have their exported symbols and symbol dereferences renamed, if necessary, so that they are inside the rump kernel namespace. Objects which do not belong to a rump kernel are not affected.

We observed that almost all of the GNU toolchain's symbols are in the double-underscore namespace "__", whereas the NetBSD kernel exported under 10 symbols in that namespace. The decision was to rename existing kernel symbols in the double underscore namespace to a single underscore namespace and exclude the double underscore namespace from the rename. There were two exceptions to the double underscore rule which had to be excluded from the rename as well: `_GLOBAL_OFFSET_TABLE_` and architecture specific ones. We handle the architecture specific ones with a quirk table. There is one quirk each for PA-RISC, MIPS, and PowerPC64. For example, the MIPS toolchain generates the symbol `_gp_disp`, which needs to be excluded from the renaming. Experience of over 5 years shows that once support for an architecture is added, no maintenance is required.

We conclude mass renaming symbols is a practical and feasible solution for the symbol collision problem which, unlike manual renaming, does not require knowledge of the set of symbols that the application namespace exports.

3.2.2 Privileged Instructions

Kernel code dealing with for example the MMU may execute CPU instructions which are available only in privileged mode. Executing privileged instructions while in non-privileged mode should cause a trap and the host OS or VMM to take control. Typically, this trap will result in process termination.

Virtualization and CPU emulation technologies solve the problem by not executing privileged instructions on the host CPU in unprivileged mode. For example, paravirtualized Xen [3] uses hypercalls, User Mode Linux [16] does not use privileged instructions in the usermode machine dependent code, QEMU [5] handles such instructions in the machine emulator, and CPU virtualization extensions trap the handling of those instructions to the hypervisor.

In practice kernel drivers do not use privileged instructions because they are found only in the architecture specific parts of the kernel. Therefore, we can solve the problem by defining that it does not exist in our model — if there are any it is a failure in modifying the OS to support rump kernels.

3.2.3 The Hypercall Interface(s)

The hypercall interfaces allow a rump kernel to access host platform resources and integrate with the host. For example, page level memory allocation and the blocking and running of threads is accomplished via the hypercall interface. Essentially, the hypercall interface represents the *minimal* interface for running kernel drivers. As we shall see later in Section 4.2, transitively the hypercall interface is also the minimal interface for running POSIX applications. Notably, the Embassies project [25] investigated a minimal execution interface for applications, and ended up with a similar interface, thereby increasing our confidence in our result being correct.

www.dbooks.org

Historically, all hypercalls were globally implemented by a single library. This was found to be inflexible for I/O devices for two reasons:

- Different I/O devices have different requirements. Imagine the hypercalls for a network interface necessary to send and receive packets. If you imagined a PCI network interface card, the necessary hypercalls are completely different from if you imagined `/dev/tap` or netmap [50]. With a growing number of different I/O devices being supported, codifying the different requirements under a compact, fast and understandable interface was not seen to be reasonably possible.

- Not all platforms require all I/O devices to be supported. Globally pooling all hypercalls together obfuscates what is the minimal required set of hypercall functionality to run rump kernel on a given platform with a given set of I/O devices.

Therefore, there are now two types of hypercalls: main hypercalls, which are always required by every rump kernel, and I/O hypercalls, which allow given types of I/O drivers to operate. The main hypercall interface is a single, stable and versioned interface. The I/O hypercalls are private to the I/O bus or the I/O driver, and an implementation is required only if the component using the hypercall is linked into the rump kernel. The versioning of I/O hypercalls and changes to them are up to individual drivers, and for example over the course of optimizing networking packet processing performance, we made several tweaks to the hypercalls used by the virtual packet I/O drivers (Section 3.9.1). These changes provided for example the ability to deliver packets in bursts and zero-copy processing. For the remainder of this section we will describe the main hypercall interface.

Main hypercalls

For the historical reason of rump kernels initially running in userspace, the hypercall interface is called *rumpuser*; for example *rumphyper* would be a more descriptive name, but changing it now brings unnecessary hassle. The version of the hypercall revision we describe here is 17. In reality, version 17 means the second stable rendition of the interface, as during initial development the interface was changed frequently and the version was bumped often. The canonical implementation for the interface is the POSIX platform implementation, currently found from `lib/librumpuser` in the NetBSD tree. Implementations for other platforms are found from `http://repo.rumpkernel.org/`, and also from 3rd parties.

As an example of a hypercall, we consider allocating memory from the host. A hypercall is the only way that a rump kernel can allocate memory at runtime. Notably, though, in the fastpath case the hypercall is used to allocate page-sized chunks of memory, which are then dosed out by the pool and slab allocators in the rump kernel. The signature of the memory allocation hypercall is the following:

```
int rumpuser_malloc(size_t howmuch, int alignment, void **retp);
```

If a hypercall can fail, its return type is `int`, and it returns 0 for success or an error code. If a hypercall cannot fail, it is of type `void`. If successful, the memory allocation hypercall will have allocated `howmuch` bytes of memory and returns a pointer to that memory in `retp`. The pointer is guaranteed to be aligned to `alignment` bytes. For example on POSIX the implementation of this interface is a call to `posix_memalign()` or equivalent.

The header file `sys/rump/include/rump/rumpuser.h` defines the hypercall interface. All hypercalls by convention begin with the string "rumpuser". This naming

www.dbooks.org

convention prevents hypercall interface references in the rump kernel from falling under the jurisdiction of symbol renaming and hence the hypercalls are accessible from the rump kernel.

The hypercalls required to run rump kernels can be categorized into the following groups:

- **initialization**: bootstrap the hypercall layer and check that the rump kernel hypercall version matches a version supported by the hypercall implementation. This interface is called as the first thing when a rump kernel initializes.
- **memory management**: allocate aligned memory, free
- **thread management**: create and join threads, TLS access
- **synchronization routines**: mutex, read/write lock, condition variable.
- **time management**: get clock value, suspend execution of calling thread for the specified duration
- **exit**: terminate the platform. Notably, in some cases it is not possible to implement this fully, e.g. on bare metal platforms without software power control. In that case, a best effort approximation should be provided.

There are also a number of optional hypercalls, which are not strictly speaking required in all cases, but are nevertheless part of core functionality:

- **errno handling**: If system calls are to be made, the hypervisor must be able to set a host thread-specific *errno* so that the client can read it. Note: errno handling is unnecessary if the clients do not use the rump system call API.

- **putchar**: output character onto console. Being able to print console output is helpful for debugging purposes.
- **printf**: a printf-like call. see discussion below.

The Benefit of a printf-like Hypercall

The `rumpuser_dprintf()` call has the same calling convention as the NetBSD kernel `printf()` routine. It is used to write debug output onto the console, or elsewhere if the implementation so chooses. While the kernel `printf()` routine can be used to produce debug output via `rumpuser_putchar()`, the kernel printf routine in-kernel locks to synchronize with other in-kernel consumers of the same interface. These locking operations may cause the rump kernel virtual CPU context to be relinquished, which in turn may cause inaccuracies in debug prints especially when hunting racy bugs. Since the hypercall runs outside of the kernel, and will not unschedule the current rump kernel virtual CPU, we found that debugging information produced by it is much more accurate. Additionally, a hypercall can be executed without a rump kernel context. This property was invaluable when working on the low levels of the rump kernel itself, such as thread management and CPU scheduling.

3.3 Rump Kernel Entry and Exit

As we discussed in Chapter 2, a client must possess an execution context before it can successfully operate in a rump kernel. These resources consist of a rump kernel process/thread context and a virtual CPU context. The act of ensuring that these resources have been created and selected is presented as pseudocode in Figure 3.3 and available as real code in `sys/rump/librump/rumpkern/scheduler.c`. We will discuss obtaining the thread context first.

www.dbooks.org

Recall from Section 2.3 that there are two types of thread contexts: an implicit one which is dynamically created when a rump kernel is entered and a bound one which the client thread has statically set. We assume that all clients which are critical about their performance use bound threads.

The entry point `rump_schedule()`[3] starts by checking if the host thread has a bound rump kernel thread context. This check maps to consulting the host's thread local storage with a hypercall. If a value is set, it is used and the entrypoint can move to scheduling a CPU.

In case an implicit thread is required, we create one. We use the system thread `lwp0` as the bootstrap context for creating the implicit thread. Since there is only one instance of this resource, it must be locked before use. After a lock on `lwp0` has been obtained, a CPU is scheduled for it. Next, the implicit thread is created and it is given the same CPU we obtained for `lwp0`. Finally, `lwp0` is unlocked and servicing the rump kernel request can begin.

The exit point is the converse: in case we were using a bound thread, just releasing the CPU is enough. In case an implicit thread was used it must be released. Again, we need a thread context to do the work and again we use `lwp0`. A critical detail is noting the resource acquiry order which must be the same as the one used at the entry point. The CPU must be unscheduled before `lwp0` can be locked. Next, a CPU must be scheduled for `lwp0` via the normal path. Attempting to obtain `lwp0` while holding on to the CPU may lead to a deadlock.

Instead of allocating and free'ing an implicit context at every entry and exit point, respectively, a possibility is to cache them. Since we assume that all performance-conscious clients use bound threads, caching would add unwarranted complexity.

[3] `rump_schedule()` / `rump_unschedule()` are slight misnomers and for example `rump_enter()` / `rump_exit()` would be more descriptive. The interfaces are exposed to clients, so changing the names is not worth the effort anymore.

```
void
rump_schedule()
{
        struct lwp *lwp;

        if (__predict_true(lwp = get_curlwp()) != NULL) {
                rump_schedule_cpu(lwp);
        } else {
                lwp0busy();

                /* allocate & use implicit thread.  uses lwp0's cpu */
                rump_schedule_cpu(&lwp0);
                lwp = rump_lwproc_allocateimplicit();
                set_curlwp(lwp);

                lwp0rele();
        }
}

void
rump_unschedule()
{
        struct lwp *lwp = get_curlwp();

        rump_unschedule_cpu(lwp);
        if (__predict_false(is_implicit(lwp))) {
                lwp0busy();

                rump_schedule_cpu(&lwp0);
                rump_lwproc_releaseimplicit(lwp);

                lwp0rele();
                set_curlwp(NULL);
        }
}
```

Figure 3.3: **Rump kernel entry/exit pseudocode**. The entrypoint and exitpoint are `rump_schedule()` and `rump_unschedule()`, respectively. The assignment of a CPU and implicit thread context are handled here.

www.dbooks.org

3.3.1 CPU Scheduling

Recall from Section 2.3.2 that the purpose of the rump kernel CPU scheduler is to map the currently executing thread to a unique rump CPU context. In addition to doing this mapping at the entry and exit points as described above, it must also be done around potentially blocking hypercalls as well. One reason for releasing the CPU around hypercalls is because the wakeup condition for the hypercall may depend on another thread being able to run. Holding on to the CPU could lead to zero available CPUs for performing a wakeup, and the system would deadlock.

The straightforward solution is to maintain a list of free virtual CPUs: allocation is done by taking an entry off the list and releasing is done by putting it back on the list. A list works well for uniprocessor hosts. However, on a multiprocessor system with multiple threads, a global list causes cache contention and lock contention. The effects of cache contention can be seen from Figure 3.4 which compares the wall time for executing 5 million `getpid()` calls per thread per CPU. This run was done 10 times, and the standard deviation is included in the graph (if it is not visible, it is practically nonexistent). The multiprocessor run took approximately three times as long as the uniprocessor one — doubling the number of CPUs made the normalized workload slower. To optimize the multiprocessor case, we developed an improved CPU scheduling algorithm.

Improved algorithm

The purpose of a rump kernel CPU scheduling algorithm is twofold. First, it ensures that at most one thread is using the CPU at any point in time. Second, it ensures that cache coherency is upheld. We dissect the latter point further. On a physical system, when thread A relinquishes a CPU and thread B is scheduled onto the same CPU, both threads will run on the same physical CPU, and therefore all data

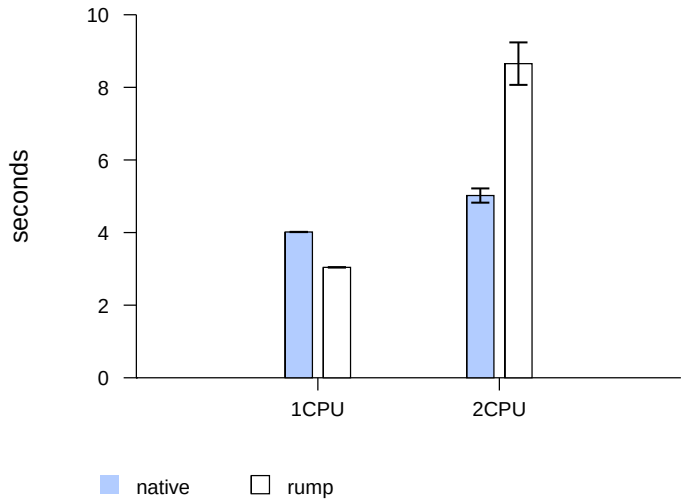


Figure 3.4: **System call performance using the trivial CPU scheduler.** While a system call into the rump kernel is faster in a single-threaded process, it is both jittery and slow for a multithreaded process. This deficiency is something we address with the advanced rump kernel CPU scheduler presented later.

they see in the CPU-local cache will trivially be coherent. In a rump kernel, when host thread A relinquishes the rump kernel virtual CPU, host thread B may acquire the same rump kernel virtual CPU on a different physical CPU. Unless the physical CPU caches are properly updated, thread B may see incorrect data. The simple way to handle cache coherency is to do a full cache update at every scheduling point. However, a full update is wasteful in the case where a host thread is continuously scheduled onto the same rump kernel virtual CPU.

The improved algorithm for CPU scheduling is presented as pseudocode in Figure 3.5. It is available as code in `sys/rump/librump/rumpkern/scheduler.c`. The scheduler is optimized for the case where the number of active worker threads is smaller than the number of configured virtual CPUs. This assumption is reasonable for rump kernels, since the amount of virtual CPUs can be configured based on each individual application scenario.

www.dbooks.org

The fastpath is taken in cases where the same thread schedules the rump kernel consecutively without any other thread running on the virtual CPU in between. The fastpath not only applies to the entry point, but also to relinquishing and rescheduling a CPU during a blocking hypercall. The implementation uses atomic operations to minimize the need for memory barriers which are required by full locks.

Next, we offer a verbose explanation of the scheduling algorithm.

1. Use atomic compare-and-swap (CAS) to check if we were the previous thread to be associated with the CPU. If that is the case, we have locked the CPU and the scheduling fastpath was successful.

2. The slow path does a full mutex lock to synchronize against another thread releasing the CPU. In addition to enabling a race-free sleeping wait, using a lock makes sure the cache of the physical CPU the thread is running on is up-to-date.

3. Mark the CPU as wanted with an atomic swap. We examine the return value and if we notice the CPU was no longer busy at that point, try to mark it busy with atomic CAS. If the CAS succeeds, we have successfully scheduled the CPU. We proceed to release the lock we took in step 2. If the CAS did not succeed, check if we want to migrate the lwp to another CPU.

4. In case the target CPU was busy and we did not choose to migrate to another CPU, wait for the CPU to be released. After we have woken up, loop and recheck if the CPU is available now. We must do a full check to prevent races against a third thread which also wanted to use the CPU.

```
void
schedule_cpu()
{
        struct lwp *lwp = curlwp;

        /* 1: fastpath */
        cpu = lwp->prevcpu;
        if (atomic_cas(cpu->prevlwp, lwp, CPU_BUSY) == lwp)
                return;

        /* 2: slowpath */
        mutex_enter(cpu->mutex);
        for (;;) {
                /* 3: signal we want the CPU */
                old = atomic_swap(cpu->prevlwp, CPU_WANTED);
                if (old != CPU_BUSY && old != CPU_WANTED) {
                        membar();
                        if (atomic_cas(cpu->prevlwp, CPU_WANTED, CPU_BUSY) == CPU_WANTED) {
                                break;
                        }
                }
                newcpu = migrate(lwp, cpu);
                if (newcpu != cpu) {
                        continue;
                }

                /* 4: wait for CPU */
                cpu->wanted++;
                cv_wait(cpu->cv, cpu->mutex);
                cpu->wanted--;
        }
        mutex_exit(cpu->mutex);
        return;
}
```

Figure 3.5: **CPU scheduling algorithm in pseudocode.** See the text for a detailed description.

www.dbooks.org

Releasing a CPU requires the following steps. The pseudocode is presented in Figure 3.6. The fastpath is taken if no other thread wanted to take the CPU while the current thread was using it.

1. Issue a memory barrier: even if the CPU is currently not wanted, we must perform this step.

   In more detail, the problematic case is as follows. Immediately after we release the rump CPU, the same rump CPU may be acquired by another hardware thread running on another physical CPU. Although the scheduling operation must go through the slowpath, unless we issue the memory barrier before releasing the CPU, the releasing CPU may have cached data which has not reached global visibility.

2. Release the CPU with an atomic swap. The return value of the swap is used to determine if any other thread is waiting for the CPU. If there are no waiters for the CPU, the fastpath is complete.

3. If there are waiters, take the CPU lock and perform a wakeup. The lock necessary to avoid race conditions with the slow path of `schedule_cpu()`.

```
void
unschedule_cpu()
{
        struct lwp *lwp = curlwp;

        /* 1: membar */
        membar();

        /* 2: release cpu */
        old = atomic_swap(cpu->prevlwp, lwp);
        if (old == CPU_BUSY) {
                return;
        }

        /* 3: wake up waiters */
        mutex_enter(cpu->mutex);
        if (cpu->wanted)
                cv_broadcast(cpu->cv);
        mutex_exit(cpu->mutex);
        return;
}
```

Figure 3.6: **CPU release algorithm in pseudocode.** See the text for a detailed description.

www.dbooks.org

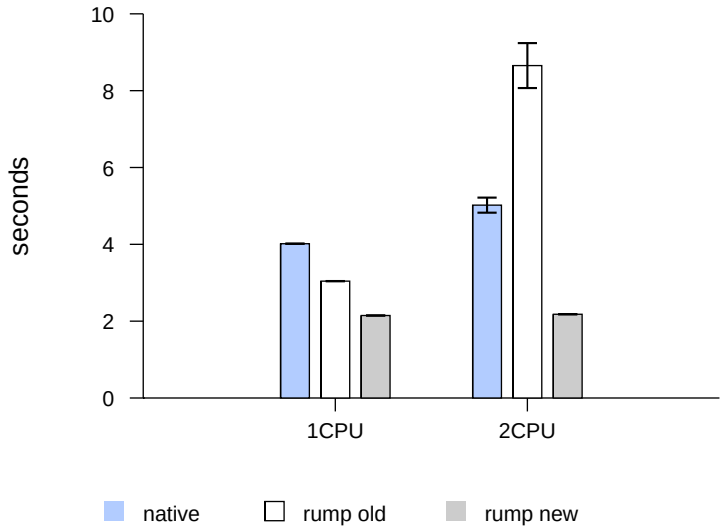


Figure 3.7: System call performance using the improved CPU scheduler. The advanced rump kernel CPU scheduler is lockless and cache conscious. With it, simultaneous system calls from multiple threads are over twice as fast as against the host kernel and over four times as fast as with the old scheduler.

Performance

The impact of the improved CPU scheduling algorithm is shown in Figure 3.7. The new algorithm performs four times as good as the freelist algorithm in the dual CPU multithreaded case. It also performs twice as fast as a host kernel system call. Also, there is scalability: the dual CPU case is within 1% of the performance of the single CPU case — native performance is 20% weaker with two CPUs. Finally, the jitter we set out to eliminate has been eliminated.

CPU-bound lwps

A CPU-bound lwp will execute only on a specific CPU. This functionality is required for example for delivering a clock interrupt on every virtual CPU. Any lwp which

is bound to a certain rump kernel virtual CPU simply has migration disabled. This way, the scheduler will always try to acquire the same CPU for the thread.

Scheduler Priorities

The assumption is that a rump kernel is configured with a number of virtual CPUs which is equal or greater to the number of frequently executing threads. Despite this configuration, a rump kernel may run into a situation where there will be competition for virtual CPUs. There are two ways to approach the issue of deciding in which order threads should be given a rump CPU context: build priority support into the rump CPU scheduler or rely on host thread priorities.

To examine the merits of having priority support in the rump CPU scheduler, we consider the following scenario. Thread A has higher priority than thread B in the rump kernel. Both are waiting for the same rump kernel virtual CPU. Even if the rump CPU scheduler denies thread B entry because the higher priority thread A is waiting for access, there is no guarantee that the host schedules thread A before thread B could theoretically run to completion in the rump kernel. By this logic, it is better to let host priorities dictate, and hand out rump kernel CPUs on a first-come-first-serve basis. Therefore, we do not support thread priorities in the rump CPU scheduler. It is the client's task to call `pthread_setschedparam()` or equivalent if it wants to set a thread's priority.

3.3.2 Interrupts and Soft Interrupts

As mentioned in Section 2.3.3, a rump kernel CPU cannot be preempted. The mechanism of how an interrupt gets delivered requires preemption, so we must examine that we meet the requirements of both hardware interrupts and *soft interrupts*.

www.dbooks.org

Hardware interrupt handlers are typically structured to only do a minimal amount of work for acknowledging the hardware. They then schedule the bulk work to be done in a soft interrupt (*softint*) handler at a time when the OS deems suitable.

As mentioned in Section 2.3.3, we implement interrupts as threads which schedule a rump kernel CPU, run the handler, and release the CPU. The only difference to a regular system is that interrupts are scheduled instead of preempting the CPU.

Softints in NetBSD are almost like regular threads. However, they have a number of special properties to keep scheduling and running them cheap:

1. Softints are run by level (e.g. networking and clock). Only one softint per level per CPU may be running, i.e. softints will run to finish before the next one may be started. Multiple outstanding softints will be queued until the currently running one has finished.

2. Softints may block briefly to acquire a short-term lock (mutex), but may not sleep. This property is a corollary of the previous property.

3. Softint handlers must run on the same CPU they were scheduled to. By default, softints are scheduled on the calling CPU. However, to distribute interrupt load, NetBSD also allows scheduling softints to other CPUs. Regardless, once the handler has been scheduled, it runs entirely on the scheduled CPU.

4. A softint may run only after the hardware interrupt finishes. That is to say, the softint handler may not run immediately after it is scheduled, only when the hardware interrupt handler that scheduled it has completed execution.

Although in a rump kernel even "hardware" interrupts are already scheduled, a fair amount of code in NetBSD assumes that softint semantics are supported. For

example, the callout framework [9] schedules soft interrupts from hardware clock interrupts to run periodic tasks (used e.g. by TCP timers).

The users of the kernel softint facility expect them to operate exactly according to the principles we listed. Initially, for simplicity, softints were implemented as regular threads. The use of regular threads resulted in a number of problems. For example, when the Ethernet code schedules a soft interrupt to do IP level processing for a received frame, code first schedules the softint and only later adds the frame to the processing queue. When softints were implemented as regular threads, the host could run the softint thread before the Ethernet interrupt handler had put the frame on the processing queue. If the softint ran before the packet was queued, the packet would not be handled until the next incoming packet.

Soft interrupts are implemented in `sys/rump/librump/rumpkern/intr.c`. The NetBSD implementation was not usable for rump kernels since that implementation is based on interaction with the NetBSD scheduler. Furthermore, the NetBSD implementation uses interprocess interrupts (IPIs) to schedule softints onto other CPUs. Rump kernels do not have interrupts or interprocessor interrupts. Instead, a helper thread is used. When scheduling a softint onto another rump kernel CPU, the helper thread schedules itself onto that virtual CPU and schedules the softint like for a local CPU. While that approach is not as performant as using IPIs, our assumption is that in high-performance computing the hardware interrupt is already scheduled onto the CPU where the work should be handled, thereby making the cost of scheduling a softint onto another CPU a non-issue.

3.4 Virtual Memory Subsystem

The main purpose of the NetBSD virtual memory subsystem is to manage memory address spaces and the mappings to the backing content [10]. While the memory

www.dbooks.org

address spaces of a rump kernel and its clients are managed by their respective hosts, the virtual memory subsystem is conceptually exposed throughout the kernel. For example, file systems are tightly built around being able to use virtual memory subsystem data structures to cache file data. To illustrate, consider the standard way the kernel reads data from a file system: memory map the file, access the mapped range, and possibly fault in missing data [51].

Due to the design choice that a rump kernel does not use (nor require) a hardware MMU, the virtual memory subsystem implementation is different from the regular NetBSD VM. As already explained in Section 2.4, the most fundamental difference is that there is no concept of page protection or a page fault inside the rump kernel.

The details of the rump kernel VM implementation along with their implications are described in the following subsections. The VM is implemented in the source module `sys/rump/librump/rumpkern/vm.c`. Additionally, routines used purely by the file system faction are in `sys/rump/librump/rumpvfs/vm_vfs.c`.

Pages

When running on hardware, the pages described by the `struct vmpage` data structure correspond with hardware pages[4]. Since the rump kernel does not interface with the MMU, the size of the memory page is merely a programmatical construct: the kernel hands out *physical* memory in multiples of the page size. In a rump kernel this memory is allocated from the host and since there is no memory protection or faults, the page size can in practice be any power of two within a sensible size range. However, so far there has been no reason to use anything different than the page size for the machine architecture the rump kernel is running on.

[4] This correspondence is not a strict rule. For example the NetBSD VAX port uses clusters of 512 byte contiguous hardware pages to create logical 4kB pages to minimize management overhead.

The VM tracks status of when a page was last used. It does this tracking either by asking the MMU on CPU architectures where that is supported, e.g. i386, or by using memory protection and updating the information during page faults on architectures where it is not, e.g. alpha. This information is used by the page daemon during memory shortages to decide which pages are best suited to be paged to secondary storage so that memory can be reclaimed for other purposes. Instead of requiring a MMU to keep track of page usage, we observe that since memory pages allocated from a rump kernel cannot be mapped into a client's address space, the pages are used only in kernel code. Every time kernel code wants to access a page, it does a lookup for it using `uvm_pagelookup()`, uses it, and releases the reference. Therefore, we hook usage information tracking to the lookup routine: whenever a lookup is done, the page is deemed as accessed.

3.4.1 Page Remapping

In practice, the kernel does not map physical pages in driver code. However, there is one exception we are interested in: the file system independent vnode pager. We will explain the situation in detail. The pages associated with a vnode object are cached in memory at arbitrary memory locations [51]. Consider a file which is the size of three memory pages. The content for file offset `0x0000-0x0FFF` might be in page `X`, `0x1000-0x1FFF` in page `X-1` and `0x2000-0x2FFF` in page `X+1`. In other words, reading and writing a file is a scatter-gather operation with respect to memory addresses. When the standard vnode pager (`sys/miscfs/genfs/genfs_io.c`) writes contents from memory to backing storage, it first maps all the pages belonging to the appropriate offsets in a continuous memory address by calling `uvm_pagermapin()`. This routine in turn uses the *pmap* interface to request the MMU to map the physical pages to the specified virtual memory range in the kernel's address space. After this step, the vnode pager performs I/O on this *pager window*. When I/O is complete, the pager window is unmapped. Reading works essentially the same way: pages

www.dbooks.org

are allocated, mapped into a contiguous window, I/O is performed, and the pager window is unmapped.

To support the standard NetBSD vnode pager with its remapping feature, there are three options for dealing with `uvm_pagermapin()`:

1. Create the window by allocating a new block of contiguous anonymous memory and use memory copy to move the contents. This approach works because pages are unavailable to other consumers during I/O; otherwise e.g. `write()` at an inopportune time might cause a cache flush to write half old half new contents and cause a semantic break.
2. Modify the vnode pager to issue multiple I/O requests in case the backing pages for a vnode object are not at consecutive addresses.
3. Accept that memory remapping support is necessary in a rump kernel.

It should be noted that a fourth option is to implement a separate vnode pager which does not rely on mapping pages. This option was our initial approach. While the effort produced a superficially working result, we could not get all corner cases to function exactly the same as with the regular kernel — for example, the `VOP_GETPAGES()` interface implemented by the vnode pager takes 8 different parameters and 14 different flags. The lesson learnt from this attempt with the vnode pager reflects our premise for the entire work: it is easy to write superficially working code, but getting all corner cases right for complicated drivers is extremely difficult.

So, which of the three options is the best? When comparing the first and the second option, the principle used is that memory I/O is several orders of magnitude faster than device I/O. Therefore, anything which affects device I/O should be avoided, especially if it might cause extra I/O operations and thus option 1 is preferable over option 2.

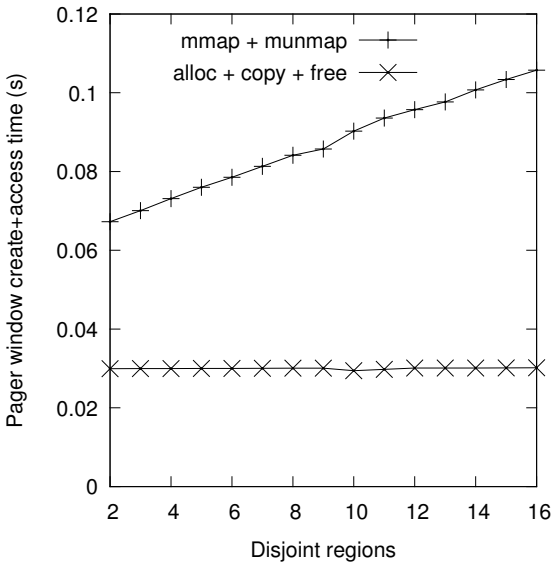


Figure 3.8: **Performance of page remapping vs. copying.** Allocating a pager window from anonymous memory and copying file pages to it for the purpose of pageout by the vnode pager is faster than remapping memory backed by a file. Additionally, the cost of copying is practically independent of the amount of non-contiguous pages. With remapping, each disjoint region requires a separate call to `mmap()`.

To evaluate the first option against third option, let us ignore MMU-less environments where page remapping is not possible, and consider an environment where it is possible albeit clumsy: userspace. Usermode operating systems typically use a memory mapped file to represent the physical memory of the virtual kernel [16, 17]. The file acts as a handle and can be mapped to the location(s) desired by the usermode kernel using the `mmap()` system call. The DragonFly usermode *vkernel* uses special host system calls to make the host kernel execute low level mappings [17].

We simulated pager conditions and measured the amount of time it takes to construct a contiguous 64kB memory window out of non-contiguous 4kB pages and to write the window out to a file backed by a memory file system. The result for 1000 loops as a function of non-contiguous pages is presented in Figure 3.8.

www.dbooks.org

The conclusion is that without direct access to a MMU, page remapping is either slower than memory copy or impossible. The downside of memory copy is that in low-memory environments you need twice the amount of memory of the largest allowed pager window size. Furthermore, the pagedaemon (Section 3.4.3) needs the window's worth of reserved memory to ensure that it is possible to flush out pages and release memory. The simple choice is to mitigate the problems by restricting the pager window size.

3.4.2 Memory Allocators

Although memory allocators are not strictly speaking part of the virtual memory subsystem, they are related to memory so we describe them here.

The lowest level memory allocator in NetBSD is the UVM kernel memory allocator (`uvm_km`). It is used to allocate memory on a pagelevel granularity. The standard implementation in `sys/uvm/uvm_km.c` allocates a virtual memory address range and, if requested, allocates physical memory for the range and maps it in. Since mapping is incompatible with a rump kernel, we did a straightforward implementation which allocates a page or contiguous page range with a hypercall.

The *kmem*, *pool* and *pool_cache* allocators are general purpose allocators meant to be used by drivers. Fundamentally, they operate by requesting pages from `uvm_km` and handing memory out in requested size chunks. The flow of memory between UVM and the allocators is dynamic, meaning if an allocator runs out of memory, it will request more from UVM, and if there is a global memory shortage, the system will attempt to reclaim cached memory from the allocators. We have extracted the implementations for these allocators from the standard NetBSD kernel and provide them as part of the rump kernel base.

3.4.3 Pagedaemon

The NetBSD kernel uses idle memory for caching data. As long as free memory is available, it will be used for caching. NetBSD's pagedaemon serves the purpose of pushing out unnecessary data to recycle pages when memory is scarce. A mechanism is required to keep long-running rump kernels from consuming all memory for caching. The choices are to either eliminate caching and free memory immediately after it has been used, or to create a pagedaemon which can operate despite memory access information not being available with the help of a MMU. Since eliminating caching is undesirable for performance reasons, we chose the latter option.

Typically, a system will have a specific amount of memory assigned to it. A straightforward example is a system running directly on hardware. While we could always require that the amount of memory be specified, that would introduce a default and configurable which is not always necessary. Since host memory is dynamically allocated using hypercalls, we can observe that in some cases we simply do not have to configure the amount of available memory. For example, short-lived test cases for kernel drivers running on a userspace host do not need one. For the rest of the discussion, we do assume that we have an "inflexible" use case and host, and do need to configure the amount of available memory.

When the available memory is close to being exhausted, the rump kernel invokes the pagedaemon, which is essentially a kernel thread, to locate and free memory resources which are least likely to be used in the near future. There are fundamentally two types of memory: pageable and wired.

- **Pageable memory** means that a memory page can be paged out. Paging is done using the pager construct that the NetBSD VM (UVM) inherited from the Mach VM [49] via the 4.4BSD VM. A pager has the capability to move the contents of the page in and out of secondary storage. NetBSD

www.dbooks.org

currently supports three classes of pagers: anonymous, vnode and device. Device pagers map device memory, so they can be left out of a discussion concerning RAM. We extract the standard UVM anonymous memory object implementation (`sys/uvm/uvm_aobj.c`) mainly because the tmpfs file system requires anonymous memory objects. However, we compile `uvm_aobj.c` without defining `VMSWAP`, i.e. the code for support moving memory to and from secondary is not included. Our view is that paging anonymous memory should be handled by the host. What is left is the vnode pager, i.e. moving file contents between the memory cache and the file system.

- **Wired memory** is non-pageable, i.e. it is always present and mapped. Still, it needs to be noted that the *host* can page memory which is wired in the rump kernel barring precautions such as a hypercall invoking `mlock()` — DMA-safe memory notwithstanding (Section 4.2.5), this paging has no impact on the rump kernel's correctness. During memory shortage, the pagedaemon requests the allocators to return unused pages back to the system.

The pagedaemon is implemented in the `uvm_pageout()` routine in the source file `sys/rump/librump/rumpkern/vm.c`. The pagedaemon is invoked when memory use exceeds the critical threshold, and additionally when the memory allocation hypercall fails. The pagedaemon releases memory in stages, from the ones most likely to bring benefit to the least likely. The use case the pagedaemon was developed against was the ability to run file systems with a rump kernel with limited memory. Measurements showing how memory capacity affects file system performance are presented in Table 3.2.

Since all pages managed by the VM are dynamically allocated and free'd, shrinking the virtual kernel or allowing it to allocate more memory is trivial. It is done by adjusting the limit. Making the limit larger causes the pagedaemon to cease activity until future allocations cause the new limit to be reached. Making the limit

rump kernel memory limit	relative performance
0.5MB	50%
1MB	90%
3MB	100%
unlimited (host container limit)	100%

Table 3.2: **File system I/O performance vs. available memory.** If memory is extremely tight, the performance of the I/O system suffers. A few megabytes of rump kernel memory was enough to allow file I/O processing at full media speed.

smaller causes the pagedaemon to clear out cached memory until the smaller limit is satisfied. In contrast to the ballooning technique [56], a rump kernel will fully release pages and associated metadata when memory is returned to the host.

Multiprocessor Considerations for the Pagedaemon

A rump kernel is more susceptible than a regular kernel to a single object using a majority of the available memory, if not all. This phenomenon exists because in a rump kernel it is a common scenario to use only one VM object at a time, e.g. a single file is being written/read via a rump kernel. In a regular kernel there minimally are at least a small number of active files due to various daemons and system processes running.

Having all memory consumed by a single object leads to the following scenario on a multiprocessor rump kernel:

1. A consumer running on CPU1 allocates memory and reaches the pagedaemon wakeup limit.

www.dbooks.org

2. The pagedaemon starts running on CPU2 and tries to free pages.

3. The consumer on CPU1 consumes all available memory for a single VM object and must go to sleep to wait for more memory. It is still scheduled on CPU1 and has not yet relinquished the memory object lock it is holding.

4. The pagedaemon tries to lock the object that is consuming all memory. However, since the consumer is still holding the lock, the pagedaemon is unable to acquire it. Since there are no other places to free memory from, the pagedaemon can only go to a timed sleep and hope that memory and/or unlocked resources are available when it wakes up.

This scenario killed performance, since all activity stalled at regular intervals while the pagedaemon went to sleep to await the consumer going to sleep. Notably, in a virtual uniprocessor setup the above mentioned scenario did not occur, since after waking up the pagedaemon the consumer would run until it got to sleep. When the pagedaemon got scheduled on the CPU and started running, the object lock had already been released and the pagedaemon could proceed to free pages. To remedy the problem in virtual multiprocessor setups, we implemented a check to see if the object lock holder is running on another virtual CPU. If the pagedaemon was unable to free memory, but it detects an object lock holder running on another CPU, the pagedaemon thread yields. This yield usually gives the consumer a chance to release the object lock so that the pagedaemon can proceed to free memory without a full sleep like it would otherwise do in a deadlock situation.

3.5 Synchronization

The NetBSD kernel synchronization primitives are modeled after the ones from Solaris [33]. Examples include mutexes, read/write locks and condition variables.

Regardless of the type, all of them have the same basic idea: a condition is checked for and if it is not met, the calling thread is put to sleep. Later, when another thread has satisfied the condition, the sleeping thread is woken up.

The case we are interested in is when the thread checking the condition blocks. In a regular kernel when the condition is not met, the calling thread is put on the scheduler's sleep queue and another thread is scheduled. Since a rump kernel is not in control of thread scheduling, it cannot schedule another thread if one blocks. When a rump kernel deems a thread to be unrunnable, it has two options: 1) spin until the host decides to schedule another rump kernel thread 2) notify the host that the current thread is unrunnable until otherwise announced.

Option 2 is desirable since it saves resources. However, no such standard interface for implementing it exists on for example a POSIX host. The closest option would be to suspend for an arbitrary period (yield, sleep, etc.). To solve the problem, we define a set of hypercall interfaces which provide the mutex, read/write lock and condition variable primitives. On for example POSIX hosts, the implementations of those hypercalls are simply thin pass-through layers to the underlying locking primitives (`pthread_mutex_lock()` etc.). Where the rump kernel interfaces do not map 1:1, such as with the `msleep()` interface, we emulate the correct behavior using the hypercall interfaces (`sys/rump/librump/rumpkern/ltsleep.c`).

As usual, for a blocking hypercall we need to unschedule and reschedule the rump kernel virtual CPU context. For condition variables making the decision to unschedule is straightforward, since we know the wait routine is going to block, and we can always release the CPU before the hypervisor calls libpthread. With some underlying locking primitives (e.g. pthread), for mutexes and read/write locks we do not know a priori if we are going to block. However, in those cases we can make a logical guess: code should be architectured to minimize lock contention, and therefore not blocking should be a more common operation than blocking. We

www.dbooks.org

first call the *try* variant of the lock operation. It does a non-blocking attempt and returns true or false depending on if the lock was taken or not. In case the lock was taken, we can return directly. If not, we unschedule the rump kernel CPU and call the blocking variant. When the blocking variant returns, perhaps immediately in a multiprocessor rump kernel, we reschedule a rump kernel CPU and return from the hypercall.

3.5.1 Passive Serialization Techniques

Passive serialization [23] is essentially a variation of a reader-writer lock where the read side of the lock is cheap and the write side of the lock is expensive, i.e. the lock is optimized for readers. It is called passive serialization because readers do not take an atomic hardware level lock. The lack of a read-side lock is made up for by deferred garbage collection, where an old copy is released only after it has reached a *quiescent state*, i.e. there are no readers accessing the old copy. In an operating system kernel the quiescent state is usually established by making the old copy unreachable and waiting until all CPUs in the system have run code.

An example of passive serialization used for example in the Linux kernel is the *read-copy update* (RCU) facility [35]. However, the algorithm is patented and can be freely implemented only in GPL or LGPL licensed code. Both licenses are seen as too restrictive for the NetBSD kernel and are not allowed by the project. Therefore, RCU itself cannot be implemented in NetBSD. Another example of passive serialization is the *rmlock* (read-mostly lock) facility offered by FreeBSD. It is essentially a reader/writer locking facility with a lockless fastpath for readers. The write locks are expensive and require cross calling other CPUs and running code on them.

One example of where NetBSD uses passive synchronization is in the loading and unloading of system calls in `sys/kern/kern_syscall.c`. These operations require

atomic locking so as to make sure no system call is loaded more than once, and also to make sure a system call is not unloaded while it is still in use. Having to take a regular lock every time a system call is executed would be wasteful, given that unloading of system calls during runtime takes place relatively seldom, if ever. Instead, the implementation uses a passive synchronization algorithm where a lock is used only for operations which are not performance-critical. We describe the elements of the synchronization part of the algorithm, and then explain how it works in a rump kernel.

Four cases must be handled:

1. execution of a system call which is loaded and functional (fast path)
2. loading a system call
3. attempting to execute an absent system call
4. unloading a system call

1: Regular Execution

Executing a system call is considered a read side lock. The essential steps are:

1. Set currently executing system call in `curlwp->l_sysent`. This step is executed lockless and without memory barriers.
2. Execute system call.
3. Clear `curlwp->l_sysent`.

www.dbooks.org

2: Loading a System Call

Modifying the syscall vector is serialized using a lock. Since modification happens seldom compared to syscall execution, this is not a performance issue.

1. Take the kernel configuration lock.
2. Check that the system call handler was not loading before we got the lock. If it was, another thread raced us into loading the call and we abort.
3. Patch the new value to the system call vector.
4. Release the configuration lock.

3: Absent System Call

NetBSD supports autoloading absent system calls. This means that when a process makes a system call that is not supported, loading a handler may be automatically attempted. If loading a handler is successful, the system call may be able to complete without returning an error to the caller. System calls which may be loaded at runtime are set to the following stub in the syscall vector:

1. Take the kernel configuration lock. Locking is not a performance problem, since any unloaded system calls will not be frequently used by applications, and therefore will not affect system performance.
2. Check that the system call handler was not loading before we got the lock. If it was, another thread raced us into loading the call and we restart handling. Otherwise, we attempt to load the system call and patch the syscall vector.
3. Release the configuration lock.

```
/*
 * Run a cross call to cycle through all CPUs.  This does two
 * things: lock activity provides a barrier and makes our update
 * of sy_call visible to all CPUs, and upon return we can be sure
 * that we see pertinent values of l_sysent posted by remote CPUs.
 */
where = xc_broadcast(0, (xcfunc_t)nullop, NULL, NULL);
xc_wait(where);
```

Figure 3.9: **Using CPU cross calls when checking for syscall users.**

4. If the system call handler was loaded (by us or another thread), restart system call handling. Otherwise, return `ENOSYS` and, due to Unix semantics, post `SIGSYS`.

4: Unloading a System Call

Finally, we discuss the most interesting case for passive serialization: the unloading of a system call. It showcases the technique that is used to avoid read-side locking.

1. Take the configuration lock.
2. Replace the system call with the stub in the system call vector. Once this operation reaches the visibility of other CPUs, the handler can no longer be called. Autoloading is prevented because we hold the configuration lock.
3. Call a cross-CPU broadcast routine to make sure all CPUs see the update (Figure 3.9, especially the comment) and wait for the crosscall to run on all CPUs. This crosscall is the key to the algorithm. There is no difference in execution between a rump kernel with virtual CPUs and a regular kernel with physical CPUs.

www.dbooks.org

4. Check if there are any users of the system call by looping over all thread soft contexts and checking `l_sysent`. If we see no instances of the system call we want to unload, we can now be sure there are no users. Notably, if we do see a non-zero amount of users, they may or may not still be inside the system call at the time of examination.

5. In case we saw threads inside the system call, prepare to return `EBUSY`: unroll step 2 by reinstating the handler in the system call vector. Otherwise, unload the system call.

6. Release the configuration lock and return success or an error code.

Discussion

The above example for system calls is not the only example of passive serialization in a rump kernel. It is also used for example to reap threads executing in a rump kernel when a remote client calls *exec* (`sys/rump/librump/rumpkern/rump.c`). Nevertheless, we wanted to describe a usage which existed independently of rump kernels.

In conclusion, passive synchronization techniques work in a rump kernel. There is no reason we would not expect them to work. For example, RCU works in a userspace environment [14] (a more easily obtained description is available in "Paper 3" here [13]). In fact, the best performing userspace implementation is one which requires threads to inform the RCU manager when they enter a quiescent state where they will not use any RCU-related resources. Since a rump kernel has a CPU model, this quiescent state reached when there has been scheduler activity on all rump kernel CPUs. In the syscall example this was accomplished by running the CPU crosscall (Figure 3.9). Therefore, no modification is required as opposed to what is required for pure userspace applications to support the quiescence based RCU userspace approach [14].

3.5.2 Spinlocks on a Uniprocessor Rump Kernel

In a non-preemptive uniprocessor kernel there is no need to take memory bus level atomic locks since nonexistent CPUs cannot race into a lock. The only thing the kernel needs to do is make sure interrupts or preemption do not occur in critical sections. Recall, there is no thread preemption in a rump kernel. While other physical CPUs may exist on the host, the rump kernel scheduler will let only one thread access the rump kernel at a time. Hence, for example the mutex lock fastpath becomes a simple variable assignment without involving the memory bus. As we mentioned already earlier, locking a non-taken lock is the code path we want to optimize, as the assumption is that lock contention should be low in properly structured code. Only in the case the mutex is locked must a hypercall be made to arrange for a sleep while waiting for the lock to be released.

We implemented alternative uniprocessor optimized locking for rump kernels in the file `sys/rump/librump/rumpkern/locks_up.c` [5]. This implementation can be used only in rump kernels with a single virtual CPU. As explained above, this implementation does not use the synchronization hypercalls unless it needs to arrange for a thread to sleep while waiting for a lock to be released.

To see how effective uniprocessor-only locking is, we measured the performance of a program which creates 200,000 files on the NetBSD tmpfs memory file system. The results are presented in Figure 3.10. Next, we analyze the results.

[5] "up" stands for uniprocessor.

www.dbooks.org

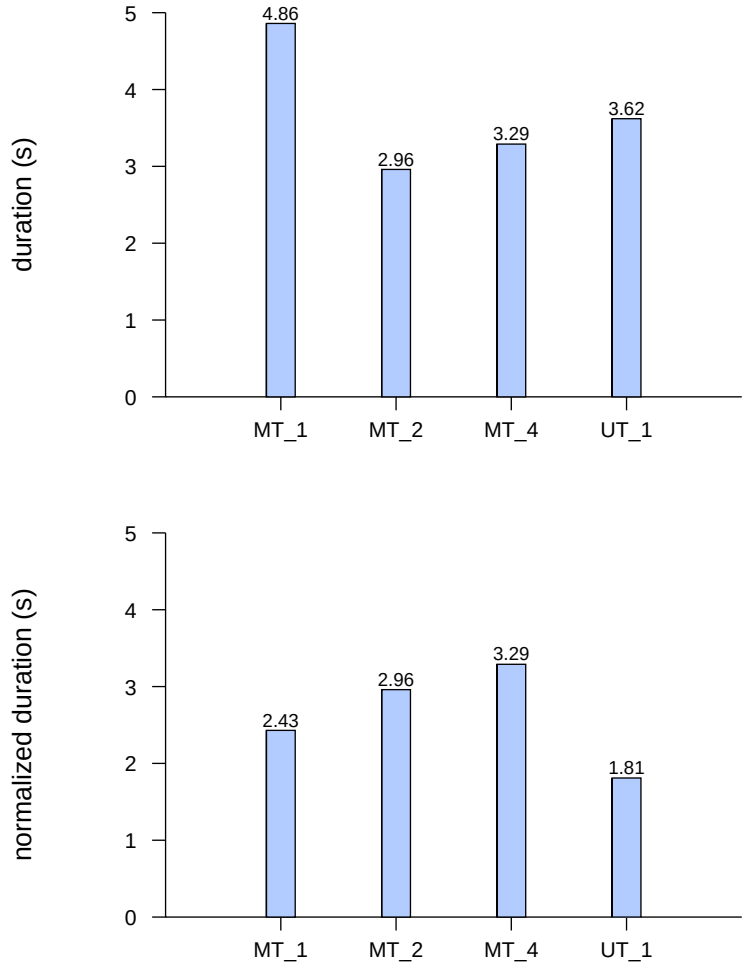


Figure 3.10: **Cost of atomic memory bus locks on a twin core host.** The first figure presents the raw measurements and the second figure presents the normalized durations per physical processor. *MT* means a multiprocessor rump kernel with hardware atomic locks and *UT* designates a uniprocessor rump kernel without hardware atomic locks. The number designates the amount of threads concurrently executing within the rump kernel. Notably, in the case of four threads there are twice as many threads executing within the rump kernel as there are physical CPUs.

The kernel with uniprocessor locking performs 34% better than the multiprocessor version on a uniprocessor rump kernel. This significant difference can be explained by the fact that creating files on memory file systems (`rump_sys_open(O_CREAT)`) is very much involved with taking and releasing locks (such as file descriptor locks, directory locks, file object locks ...) and very little involved with I/O or hypervisor calls. To verify our results, we examined the number of mutex locks and reader/writer locks and we found out they are taken 5,438,997 and 1,393,596 times, respectively. This measurement implies the spinlock/release cycle fastpath in the 100ns range, which is what we would expect from a Core2 CPU on which the test was run. The MT_4 case is slower than MT_2, because the test host has only two physical cores, and four threads need to compete for the same physical cores.

The multiprocessor version where the number of threads and virtual CPUs matches the host CPU allocation wins in wall time. However, if it is possible to distribute work in single processor kernels on all host CPUs, they will win in total performance due to IPC overhead being smaller than memory bus locking overhead [4].

3.6 Application Interfaces to the Rump Kernel

Application interfaces are used by clients to request services from the rump kernel. Having the interfaces provided as part of the rump kernel framework has two purposes: 1) it provides a C level prototype for the client 2) it wraps execution around the rump kernel entry and exit points, i.e. thread context management and rump kernel virtual CPU scheduling.

The set of available interfaces depends on the type of the client. Since the rump kernel provides a security model for remote clients, they are restricted to the system call interface — the system call interface readily checks the appropriate permissions of a caller. A local client and a microkernel server's local client are free to call

www.dbooks.org

any functions they desire. We demonstrated the ability to call arbitrary kernel interfaces with the example on how to access the BPF driver without going through the file system (Figure 2.3). In that example we had to provide our own prototype and execute the entry point manually, since we did not use predefined application interfaces.

3.6.1 System Calls

On a regular NetBSD system, a user process calls the kernel through a stub in libc. The libc stub's task is to trap into the kernel. The kernel examines the *trapframe* to see which system call was requested and proceeds to call the system call handler. After the call returns from the kernel, the libc stub sets `errno`.

We are interested in preserving the standard libc application interface signature for rump kernel clients. Preserving the signature will make using existing code in rump kernel clients easier, since the calling convention for system calls will remain the same. In this section we will examine how to generate handlers for rump kernels with minimal manual labor. All of our discussion is written against how system calls are implemented in NetBSD. We use `lseek()` as an example of the problem and our solution.

The signature of the `lseek()` system call stub in libc is as follows:

```
off_t
lseek(int fildes, off_t offset, int whence)
```

Prototypes are provided in header files. The header file varies from call to call. For example, the prototype of `lseek()` is made available to an application by including

the header file <unistd.h> while open() comes from <fcntl.h>. The system call prototypes provided in the header files are handwritten. In other words, they are not autogenerated. On the other hand, almost all libc stubs are autogenerated from a list of system calls. There are some manually written exceptions for calls which do not fit the standard mould, e.g. fork(). Since the caller of the libc stub arranges arguments according to the platform's calling convention per the supplied prototype and the kernel picks them up directly from the trapframe, the libc stub in principle has to only execute the trap instruction to initiate the handling of the system call.

In contrast to the libc application interface, the signature of the kernel entry point for the handler of the lseek system call is:

```
int
sys_lseek(struct lwp *l, const struct sys_lseek_args *uap, register_t *rv)
```

This function is called by the kernel trap handler after it has copied parameters from the trapframe to the args structure.

Native system calls are described by a master file in kernel source tree located at sys/kern/syscalls.master. The script sys/kern/makesyscalls.sh uses the data file to autogenerate, among other things, the above prototype for the in-kernel implementation and the definition of the args structure.

We added support to the makesyscalls script for generating the necessary wrappers and headers for rump kernel clients. For a caller to be able to distinguish between a native system call and a rump kernel system call, the latter is exported with a rump_sys-prefix, e.g. rump_sys_lseek(). The makesyscalls script generates rump system call prototypes to sys/rump/include/rump/rump_syscalls.h. A wrapper which takes care of arranging the function parameters into the args structure

www.dbooks.org

```
off_t
rump___sysimpl_lseek(int fd, off_t offset, int whence)
{
        register_t retval[2] = {0, 0};
        int error = 0;
        off_t rv = -1;
        struct sys_lseek_args callarg;

        SPARG(&callarg, fd) = fd;
        SPARG(&callarg, PAD) = 0;
        SPARG(&callarg, offset) = offset;
        SPARG(&callarg, whence) = whence;

        error = rsys_syscall(SYS_lseek, &callarg, sizeof(callarg), retval);
        rsys_seterrno(error);
        if (error == 0) {
                if (sizeof(off_t) > sizeof(register_t))
                        rv = *(off_t *)retval;
                else
                        rv = *retval;
        }
        return rv;
}
```

Figure 3.11: **Call stub for `rump_sys_lseek()`.** The arguments from the client are marshalled into the argument structure which is supplied to the kernel entry point. The execution of the system call is requested using the `rsys_syscall()` routine. This routine invokes either a direct function call into the rump kernel or a remote request, depending on if the rump kernel is local or remote, respectively.

is generated into `sys/rump/librump/rumpkern/rump_syscalls.c` — in our example this arranging means moving the arguments that `rump_sys_lseek()` was called with into the fields of `struct sys_lseek_args`. The wrapper for lseek is presented in Figure 3.11. The name of the wrapper in the illustration does not match `rump_sys_lseek()` but the reference will be correctly translated by an alias in the rump system call header. We will not go into details, except to say that the reason for it is to support compatibility system calls. For interested parties, the details are available in the `rump_syscalls.h` header file.

The same wrapper works both for local and remote clients. For a local client, `rsys_syscall()` does a function call into the rump kernel, while for a remote client it invokes a remote procedure call so as to call the rump kernel. Remote clients are discussed in more detail in Section 3.12. In both cases, the implementation behind `rsys_syscall()` calls the rump kernel entry and exit routines.

While modifying the makesyscalls script to generate prototypes and wrappers, we ran into a number of unexpected cases:

1. Almost all system calls return -1 (or `NULL`) in case of an error and set the `errno` variable to indicate which error happened. However, there are exceptions. For example, the `posix_fadvise()` call is specified to return an error number and not to adjust `errno`. In libc this discrepancy between error variable conventions is handled by a field in the Makefile which autogenerates syscall stubs. For our purposes of autogeneration, we added a `NOERR` flag to syscalls.master. This flag causes the generator to create a stub which does not set `errno`, much like what the libc build process does.

2. Some existing software looks only at `errno` instead of the system call's return value. Our initial implementation set `errno` only in case the system call returned failure. This implementation caused such software to not function properly and we adjusted `errno` to always be set to reflect the value from the latest call.

3. System calls return three values from the kernel: an integer and an array containing two register-size values (the `register_t *rv` parameter). In the typical case, the integer carries `errno` and `rv[0]` carries the return value. In almost all cases the second element of the register vector can be ignored. The first exception to this rule is the system call `pipe(int fildes[2])`, which returns two file descriptors from the kernel: one in `rv[0]` and the other in `rv[1]`. We handle `pipe()` as a special case in the generator script.

www.dbooks.org

```
[ .... ]
2194:       e8 fc ff ff ff          call   2195 <rump___sysimpl_lseek+0x52>
2199:       85 db                   test   %ebx,%ebx
219b:       75 0c                   jne    21a9 <rump___sysimpl_lseek+0x66>
219d:       8b 45 f4                mov    0xfffffff4(%ebp),%eax
21a0:       8b 55 f8                mov    0xfffffff8(%ebp),%edx
21a3:       83 c4 24                add    $0x24,%esp
21a6:       5b                      pop    %ebx
21a7:       5d                      pop    %ebp
21a8:       c3                      ret
[ .... ]
```

Figure 3.12: **Compile-time optimized `sizeof()` check.** The assembly of the generated code compiled for i386 is presented.

4. The second exception to the above is the `lseek()` call on 32bit architectures. The call returns a 64bit `off_t` [6] with the low bits occupying one register and the high bits the other one. Since NetBSD supports all combinations of 32bit, 64bit, little endian and big endian architectures, care had to be taken to have the translation from a two-element `register_t` vector to a variable work for all calls on all architectures. We use a compile-time check for data type sizes and typecast accordingly. To see why the check is required, consider the following. If the typecast is never done, lseek breaks on 32bit architectures. If the typecast to the return type is done for all calls, system calls returning an integer break on 64bit big-endian architectures.

   The above is not the only way to solve the problem. The makesyscalls.sh script detects 64bit return values and sets the `SYCALL_RET_64` flag in a system call's description. We could have hooked into the facility and created a special wrapper for lseek without the "`if (sizeof())`" clause. The compiled code is the same for both approaches (Figure 3.12), so the choice is a matter of taste instead of runtime performance.

[6] `off_t` is always 64bit on NetBSD instead of depending on the value of `_FILE_OFFSET_BITS` which used on for example Linux and Solaris.

5. Some calling conventions (e.g. ARM EABI) require 64bit parameters to be passed in even numbered registers. For example, consider the lseek call. The first parameter is an integer and is passed to the system call in register 0. The second parameter is 64bit, and according to the ABI it needs to be passed in registers 2+3 instead of registers 1+2. To ensure the alignment constraint matches in the kernel, the system call description table syscalls.master contains padding parameters. For example, lseek is defined as `lseek(int fd, int pad, off_t offset, int whence)`. Since "pad" is not a part of the application API, we do not want to include it in the rump kernel system call signature. However, we must include padding in the `struct sys_lseek_args` parameter which is passed to the kernel. We solved the issue by first renaming all pad parameters to the uppercase "PAD" to decrease the possibility of conflict with an actual parameter called "pad". Then, we modified makesyscalls.sh to ignore all parameters named "PAD" for the application interface side.

A possibility outside of the scope of this work is to examine if the libc system call stubs and prototypes can now be autogenerated from `syscalls.master` instead of requiring separate code in the NetBSD libc Makefiles and system headers.

3.6.2 vnode Interface

The vnode interface is a kernel internal interface. The vnode interface routines take a vnode object along with other parameters, and call the respective method of the file system associated with the vnode. For example, the interface for reading is the following: `int VOP_READ(struct vnode *, struct uio *, int, kauth_cred_t)`; if the first parameter is a pointer to a FFS vnode, the call will be passed to the FFS driver.

www.dbooks.org

```
int
RUMP_VOP_READ(struct vnode *vp, struct uio *uio, int ioflag, struct kauth_cred *cred)
{
        int error;

        rump_schedule();
        error = VOP_READ(vp, uio, ioflag, cred);
        rump_unschedule();

        return error;
}
```

Figure 3.13: **Implementation of `RUMP_VOP_READ()`.** The backend kernel call is wrapped around the rump kernel entrypoint and exitpoint.

The rump vnode interface exports the vnode interfaces to rump kernel clients. The intended users are microkernel file servers which use rump kernels as backends. The benefits for exporting this interface readily are the ones we listed in the beginning of this section: a prototype for client code and automated entry/exit point handling.

The wrappers for the vnode interface are simpler than those of the system call interface. This simplicity is because there is no need translate parameters and we can simply pass them on to the kernel internal interface as such. To distinguish between the internal implementation and the rump application interface, we prefix rump client vnode interfaces with `RUMP_`.

The kernel vnode interface implementations and prototypes are autogenerated from the file `sys/kern/vnode_if.src` by `sys/kern/vnode_if.sh`. We made the script to generate our prototypes into `sys/rump/include/rump/rumpvnode_if.h` and wrapper functions into `sys/rump/librump/rumpvfs/rumpvnode_if.c`. An example result showing the `RUMP_VOP_READ()` interface is presented in Figure 3.13. The `VOP_READ()` routine called by the wrapper is the standard implementation which is extracted into a rump kernel from `sys/kern/vnode_if.c`.

```
int
rump_pub_lwproc_rfork(int arg1)
{
        int rv;

        rump_schedule();
        rv = rump_lwproc_rfork(arg1);
        rump_unschedule();

        return rv;
}
```

Figure 3.14: Application interface implementation of lwproc `rfork()`. The backend kernel call is wrapped around the rump kernel entrypoint and exitpoint.

3.6.3 Interfaces Specific to Rump Kernels

Some interfaces are available only in rump kernels, for example the lwp/process context management interfaces (manual page *rump_lwproc.3*). In a similar fashion to other interface classes we have discussed, we supply autogenerated prototypes and wrappers.

The application interface names are prefixed with `rump_pub_` (shorthand for public). The respective internal interfaces are prefixed `rump_`. As an example, we present the wrapper for `rump_pub_lwproc_rfork()` in Figure 3.14. The public interface wraps the internal interface around the entrypoint and exitpoint.

The master files for rump kernel interfaces are contained in the subdirectory of each faction in an *.ifspec* file. The script `sys/rump/librump/makeifspec.sh` analyzes this file and autogenerates the prototypes and wrappers.

Additionally, there exist bootstrap interfaces which can be called only before the rump kernel is bootstrapped. An example is `rump_boot_sethowto()` which sets the

www.dbooks.org

`boothowto` variable. Since there is no virtual CPU to schedule before bootstrap, no entry/exit wrappers are necessary. These bootstrap interfaces provided as non-generated prototypes in `sys/rump/include/rump/rump.h`.

3.7 Rump Kernel Root File System

Full operating systems require a root file system with persistent storage for files such as `/bin/ls` and `/etc/passwd`. A rump kernel does not inherently require such files. This relaxed requirement is because a rump kernel does not have a default userspace and because client binaries are executed outside of the rump kernel. However, specific drivers or clients may require file system support for example to open a device, load firmware or access a file system image. In some cases, such as for firmware files and file system images, it is likely that the backing storage for the data to be accessed resides on the host.

We explicitly want to avoid mandating the association of persistent storage with a rump kernel because the storage image requires setup and maintenance and would hinder especially one-time invocations. It is not impossible to store the file system hierarchy and data required by a specific rump kernel instance on persistent storage. We are merely saying it is not required.

A file system driver called *rumpfs* was written. It is implemented in the source module `sys/rump/librump/rumpvfs/rumpfs.c`. Like *tmpfs*, rumpfs is an in-memory file system. Unlike tmpfs, which is as fast and as complete as possible, rumpfs is as lightweight as possible. Most rumpfs operations have only simple implementations and support for advanced features such as rename and NFS export has been omitted. If these features are desired, an instance of tmpfs can be mounted within the rump kernel when required. The lightweight implementation of rumpfs makes the compiled size 3.5 times smaller than that of tmpfs.

By convention, file system device nodes are available in `/dev`. NetBSD does not feature a device file system which dynamically creates device nodes based on the drivers in the kernel. A standard installation of NetBSD relies on precreated device nodes residing on persistent storage. We work around this issue in two ways. First, during bootstrap, the rump kernel VFS faction generates a selection of common device nodes such as `/dev/zero`. Second, we added support to various driver attachments to create device driver nodes when the drivers are attached. These adjustments avoid the requirement to have persistent storage mounted on `/dev`.

3.7.1 Extra-Terrestrial File System

The Extra-Terrestrial File System (*etfs*) interface provides a rump kernel with access to files on the host. The etfs (manual page *rump_etfs.3*) interface is used to register host file mappings with rumpfs. Fundamentally, the purpose of etfs is the same as that of a *hostfs* available on most full system virtualization solutions. Unlike a hostfs, which typically mounts a directory from the host, etfs is oriented towards mapping individual files. The interface allows the registration of type and offset translators for individual host files; a feature we will look at more closely below. In addition, etfs only supports reading and writing files and cannot manipulate the directory namespace on the host. This I/O-oriented approach avoids issues such as how to map permissions on newly created hostfs files. Furthermore, it makes etfs usable also on hosts which do not support a directory namespace.

The mapping capability of etfs is hooked up to the lookup operation within rumpfs. Recall, a lookup operation for a pathname will produce an in-memory file system structure referencing the file behind that pathname. If the pathname under lookup consists of a registered etfs key, the in-memory structure will be tagged so that further I/O operations, i.e. read and write, will be directed to the backing file on the host.

www.dbooks.org

Due to how etfs is implemented as part of the file system lookup routine, the mapped filenames is not browseable (i.e. readdir). However, it does not affect the intended use cases such as access to firmware images, since the pathnames are hardcoded into the kernel.

In addition to taking a lookup key and the backing file path, the etfs interface takes an argument controlling how the mapped path is presented inside the rump kernel. The following three options are valid for non-directory host files: regular file, character device or block device. The main purpose of the type mapping feature is to be able to present a regular file on the host as a block device in the rump kernel. This mapping addresses an implementation detail in the NetBSD kernel: the only valid backends for disk file systems are block devices.

Assuming that the host supports a directory namespace, it is also possible to map directories. There are two options: a single-level mapping or the mapping of the whole directory subtree. For example, if `/rump_a` from the host is directory mapped to `/a` in the rump kernel, it is possible to access `/rump_a/b` from `/a/b` in both single-level and subtree mappings. However, `/rump_a/b/c` is visible at `/a/b/c` only if the directory subtree was mapped. Directory mappings do not allow the use of the type and offset/size translations, but allow mappings without having to explicitly add them for every single file. The original use case for the directory mapping functionality was to get the kernel module directory tree from `/stand` on the host mapped into the rump kernel namespace so that a rump kernel could read kernel module binaries from the host.

3.7.2 External Storage

Another special feature of rumpfs is the possibility to attach external storage to regular files. This external storage is provided in the form of memory, and is made

available via files in a zero-copy fashion. The intent is to allow rump kernels to provide file content without having to rely on the presence of any block I/O device. The content itself can be linked into the data segment of the binary at the time that the binary is built. External storage is attached by opening a writable file and calling `rump_sys_ioctl(fd, RUMP_FCNTL_EXTSTORAGE_ADD, ...)`. Adding external storage is limited to local clients, as pointers provided by remote clients are meaningless in this context.

3.8 Attaching Components

A rump kernel's initial configuration is defined by the components that are linked in when the rump kernel is bootstrapped. At bootstrap time, the rump kernel needs to detect which components were included in the initial configuration and attach them. If drivers are loaded at runtime, they need to be attached to the rump kernel as well.

In this section we go over how loading and attaching components in a rump kernel is similar to a regular kernel and how it is different. The host may support static linking, dynamic linking or both. We include both alternatives in the discussion. There are two methods for attaching components, called *kernel modules* and *rump components*. We will discuss both and point out the differences. We start the discussion with kernel modules.

3.8.1 Kernel Modules

In NetBSD terminology, a driver which can be loaded and unloaded at runtime is said to be *modular*. The loadable binary image containing the driver is called a *kernel module*, or *module* for short. We adopt the terminology for our discussion.

www.dbooks.org

The infrastructure for supporting modular drivers on NetBSD has been available since NetBSD 5.0 [7]. Some drivers offered by NetBSD are modular, and others are being converted. A modular driver knows how to attach to the kernel and detach from the kernel both when it is statically included and when it is loaded at runtime. A non-modular driver does not know.

NetBSD divides kernel modules into three classes depending on their source and when they are loaded. These classes are summarized in Table 3.3. Builtin modules are linked into the kernel image when the kernel is built. The bootloader can load kernel modules into memory at the same time as it loads the kernel image. These modules must later be linked by the kernel during the bootstrap process. Finally, at runtime modules must be both loaded and linked by the kernel.

source	loading	linking	initiated by
builtin	external	external	external toolchain
bootloader	external	kernel	bootloader
file system	kernel	kernel	syscall, kernel autoload

Table 3.3: **Kernel module classification.** These categories represent the types of kernel modules that were readily present in NetBSD independent of this work.

The fundamental steps of loading a kernel module on NetBSD at runtime are:

1. The kernel module is loaded into the kernel's address space.
2. The loaded code is linked with the kernel's symbol table.
3. The module's init routine is run. This routine informs other kernel subsystems that a new module is present. For example, a file system module at a

[7] NetBSD 5.0 was released in 2009. Versions prior to 5.0 provided kernel modules through a different mechanism called Loadable Kernel Modules (*LKM*). The modules available from 5.0 onward are incompatible with the old LKM scheme. The reasons why LKM was retired in favor of the new system are available from mailing list archives and beyond the scope of this document.

minimum informs the VFS layer that it is possible to mount a new type of file system.

After these steps have been performed, code from the newly loaded kernel module can be used like it had been a part of the original monolithic kernel build. Unloading a kernel module is essentially reversing the steps.

We divide loading a module into a rump kernel in two separate cases depending on a pivot point in the execution: the bootstrapping of the rump kernel by calling `rump_init()`. The officially supported way of including modules in rump kernels is to have them loaded and linked (i.e. steps 1+2) before `rump_init()` is called. This may be done either by linking them into a static image, or, on hosts where dynamic linking is supported, loading and linking the component after `main()` is called but before `rump_init()` is called. Modules loaded that way are essentially builtin modules.

We also point out that on some hosts, especially userspace, it is possible to load components after `rump_init()` by using the host dynamic linker and then calling `rump_pub_module_init()`. However, will we not discuss the latter approach in this document.

init/fini

A NetBSD kernel module defines an init routine ("`modcmd_init`") and a fini routine ("`modcmd_fini`") using the `MODULE()` macro. The indicated routines attach and detach the module with respect to the kernel. The `MODULE()` macro creates a structure (`struct modinfo`) containing that information. We need to locate the structure at runtime so that the init routine can be run to complete module loading step "3".

www.dbooks.org

The method of locating the `modinfo` structures depends on the platform. There are two functionally equivalent options, but they have different platform requirements.

1. A pointer to the structure is placed into a `.rodata` section called `modules`. At linktime, the linker generates `__start_section` and `__end_section` symbols. At runtime, all present modules can be found by traversing pointers in the memory between these two symbols. Notably, not all linkers can be coerced into generating these symbols. Furthermore, the scheme is not directly compatible with dynamic linkers because a dynamic linker cannot produce a global pair of such symbols – consider the scenario where libraries are loaded at runtime. Therefore, if this scheme is to be used with dynamic linking, each shared object must be examined separately.

2. The structure is located via `__attribute__((__constructor__))`. We cannot make any assumptions about when the constructor runs, and therefore the constructor only places the structure on a linked list. This linked list is traversed when the rump kernel boots. The constructor is a static function generated by the `MODULE()` macro. Notably, this scheme requires platform support for running constructors.

The NetBSD Kernel Linker

Using the NetBSD kernel linker means loading the module from the file system after `rump_init()` and letting the code in `sys/kern/subr_kobj.c` handle linking. This linker is included as part of the base of a rump kernel. The in-kernel linker supports only relocatable objects (with ELF, type `ET_REL`), not shared libraries.

Since linking is performed in the [rump] kernel, the [rump] kernel must be aware of the addresses of the symbols it exports. For example, for the linker to be able to sat-

isfy an unresolved symbol to `kmem_alloc()`, it must know where the implementation of `kmem_alloc()` is located in that particular instance. In a regular kernel the initial symbol table is loaded at bootstrap time by calling the `ksyms_addsyms_explicit()` or mostly equivalent `ksyms_addsyms_elf()` routine.

In the current rumpuser hypercall revision, the symbol table is populated by the `rumpuser_dl_bootstrap()` hypercall, which is always called during rump kernel bootstrap. For the next hypercall revision, the plan is to make symbol loading a just-in-time operation which is called only if the in-kernel linker is used. This change is planned because loading the symbol table – and in dynamically linked environments harvesting it for loading – is a time-consuming operating. Based on experience, the use of the in-kernel linker is a rare operation, and unconditionally populating the symbol table is therefore wasteful.

The in-kernel linker itself works the same way as in a regular kernel. Loading a module can be initiated either by a client by using the `modctl()` system call. Notably, the kernel module is loaded from the rump kernel file system namespace, so only rump kernels with the file system faction can support the in-kernel linker.

3.8.2 Modules: Supporting Standard Binaries

By a binary kernel module we mean a kernel module object file built for the regular monolithic kernel and shipped with NetBSD in `/stand/$arch/release/modules`. Support for binary kernel modules means these objects can be loaded and linked into a rump kernel and the drivers used. This support allows a rump kernel to use drivers for which source code is not available. Short of a full virtual machine (e.g. QEMU), rump kernels are the only form of virtualization in NetBSD capable of using binary kernel modules without recompilation.

www.dbooks.org

There are two requirements for using binary kernel modules to be possible. First, the kernel module must not contain any CPU instructions which cannot be executed in unprivileged mode. As we examined in Section 3.2.2, drivers do not contain privileged instructions. Second, the rump kernel and the host kernel must share the same binary interface (ABI).

In practical terms, ABI compatibility means that the rump kernel code does not provide its own headers to override system headers and therefore all the data type definitions are the same for a regular kernel and a rump kernel. Problematic scenarios arise because, mainly due to historical reasons, some architecture specific kernel interfaces are provided as macros or inline functions. This approach does not produce a clean interface boundary, as at least part of the implementation is leaked into the caller. From our perspective, this leakage means that providing an alternate interface is more difficult.

Shortly before we started investigating kernel module compatibility, some x86 CPU family headers were changed from inline/macro definitions to function interfaces by another NetBSD developer. The commit message[8] states that the change was done to avoid ABI instability issues with kernel modules. This change essentially solved our problem with inlines and macros. It also reinforced our belief that the anykernel architecture follows naturally from properly structured code.

A remaining example of macro use in an interface is the *pmap* interface. The pmap is the interface to the architecture dependent memory management features. The interface specification explicitly allows some parts of the interface to be implemented as macros. Figure 3.15 illustrates how the x86 pmap header overrides the MI function interface for `pmap_is_modified()`. To be x86 kernel ABI compatible we provide an implementation for `pmap_test_attrs()` in the rump kernel base (`sys/rump/librump/rumpkern/arch/i386/pmap_x86.c`).

[8] revision 1.146 of `sys/arch/i386/include/cpu.h`

sys/arch/x86/include/pmap.h:

```
#define pmap_is_modified(pg)            pmap_test_attrs(pg, PG_M)
```

sys/uvm/uvm_pmap.h (MI definition):

```
#if !defined(pmap_is_modified)
bool            pmap_is_modified(struct vm_page *);
#endif
```

Figure 3.15: **Comparison of `pmap_is_modified` definitions.** The definition specific to the *i386* port causes a dereference of the symbol `pmap_test_attrs()`, while for all ports which do not override the definition, `pmap_is_modified()` is used.

Due to the MD work required, the kernel module ABI support is currently restricted to the AMD64 and i386 architectures. Support for AMD64 has an additional restriction which derives from the addressing model used by kernel code on AMD64. Since most AMD64 instructions accept only 32bit immediate operands, and since an OS kernel is a relatively small piece of software, kernel code is compiled with a memory model which assumes that all symbols are within the reach of a 32bit offset. Since immediate operands are sign extended, the values are correct when the kernel is loaded in the upper 2GB of AMD64's 64bit address space [32]. This address range is not available to user processes at least on NetBSD. Instead, we used the lowest 2GB — the lowest 2GB is the same as the highest 2GB without sign-extension. As long as the rump kernel and the binary kernel module are loaded into the low 2GB, the binary kernel module can be used as part of the rump kernel.

For architectures which do not support the standard kernel ABI, we provide override machine headers under the directory `sys/rump/include/machine`. This directory is specified first in the include search path for rump kernel compilation, and therefore headers contained in there override the NetBSD MD headers. Therefore, definitions contained in the headers for that directory override the standard NetBSD definitions. This way we can override problematic definitions in machine dependent code.

www.dbooks.org

sys/arch/x86/include/cpu.h:

```
#define curlwp                  x86_curlwp()
```

sys/arch/sparc64/include/{cpu,param}.h (simplified for presentation):

```
#define curlwp          curcpu()->ci_curlwp
#define curcpu()        (((struct cpu_info *)CPUINFO_VA)->ci_self)
#define CPUINFO_VA      (KERNEND+0x018000)
#define KERNEND         0x0e0000000     /* end of kernel virtual space */
```

Figure 3.16: **Comparison of `curlwp` definitions.** The *i386* port definition results in a function symbol dereference, while the *sparc64* port definition causes a dereference to an absolute memory address.

An example of what we consider problematic is SPARC64's definition of `curlwp`, which we previously illustrated in Figure 3.16. This approach allows us to support rump kernels on all NetBSD architectures without having to write machine specific counterparts or edit the existing MD interface definitions. The only negative impact is that architectures which depend on override headers cannot use binary kernel modules and must operate with the components compiled specifically for rump kernels.

Lastly, the kernel module must be converted to the rump kernel symbol namespace (Section 3.2.1) before linking. This conversion can be done with the `objcopy` tool similar to what is done when components are built. However, using `objcopy` would require generating another copy of the same module file. Instead of the objcopy approach, we modified the module load path in `sys/kern/subr_kobj.c` to contain a call to `kobj_renamespace()` after the module has been read from storage but before it is linked. On a regular kernel this interface is implemented by a null operation, while in a rump kernel the call is implemented by a symbol renaming routine in `sys/rump/librump/rumpkern/kobj_rename.c`. Since the translation is done in memory, duplicate files are not required. Also, it enables us to autoload binary modules directly from the host, as we describe next.

3.8.3 Rump Component Init Routines

In the previous section we discussed the attachment of drivers that followed the kernel module framework. Now we discuss the runtime attachment of drivers that have not yet been converted to a kernel module, or code that applies to a only rump kernel environment.

If a driver is modular, the module's init routine should be preferred over any rump kernel specific routines since the module framework is more generic. However, a module's init routine is not always enough for a rump kernel. Consider the following cases. On a regular system, parts of the kernel are configured by userspace utilities. For example, the Internet address of the loopback interface (`127.0.0.1`) is configured by the *rc scripts* instead of by the kernel. Another example is the creation of device nodes on the file system under the directory `/dev`. NetBSD does not have a dynamic device file system and device nodes are pre-created with the `MAKEDEV` script. Since a rump kernel does not have an associated userland or a persistent root file system, these configuration actions must be performed by the rump kernel itself. A rump component init routine may be created to augment the module init routine.

By convention, we place the rump component init routines in the component's source directory in a file called `$name_component.c`, e.g. `bpf_component.c`. To define an init routine, the component should use the `RUMP_COMPONENT()` macro. The use of this macro serves the same purpose as the `MODULE()` macro and ensures that the init routine is automatically called during rump kernel bootstrap.

The `RUMP_COMPONENT()` macro takes as arguments a single parameter indicating when the component should be initialized with respect to other components. This specifier is required because of interdependencies of components that the NetBSD kernel code imposes. For example, the networking domains must be attached before

www.dbooks.org

interfaces can be configured. It is legal (and sometimes necessary) for components to define several init routines with different configuration times. The init level parameters are listed in Table 3.4 in order of runtime initialization. Notably, the multitude of networking-related initialization levels conveys the current status of the NetBSD TCP/IP stack: it is not yet modular — a modular TCP/IP stack would encode the cross-dependencies in the drivers themselves.

An example of a component file is presented in Figure 3.17. The two routines specified in the component file will be automatically executed at the appropriate times during rump kernel bootstrap so as to ensure that any dependent components have been initialized before. The full source code for the file can be found from the source tree path `sys/rump/net/lib/libnetinet/netinet_component.c`. The userspace rc script `etc/rc/network` provides the equivalent functionality in a regular monolithic NetBSD setup.

3.9 I/O Backends

I/O backends allow a rump kernel to access I/O resources on the host and beyond. Access to I/O devices always stems from hypercalls, and like we learned in Section 3.2.3, hypercalls for accessing I/O devices are specific to the bus or device. For example, accessing a PCI NIC is completely different from accessing `/dev/tap`.

In this section we will discuss the implementation possibilities and choices for the I/O backends for networking and file systems (block devices) in userspace. Other related sections are the ones which discuss USB via *ugen* (Section 3.10) and PCI on the Rumprun unikernel (Section 4.2.5). There is also support for accessing PCI devices from userspace available at `http://repo.rumpkernel.org/pci-userspace`, but we do not discuss that implementation in this book.

level	purpose
`RUMP_COMPONENT_KERN`	base initialization which is done before any factions are attached
`RUMP_COMPONENT_VFS`	VFS components
`RUMP_COMPONENT_NET`	basic networking, attaching of networking domains
`RUMP_COMPONENT_NET_ROUTE`	routing, can be done only after all domains have attached
`RUMP_COMPONENT_NET_IF`	interface creation (e.g. `lo0`)
`RUMP_COMPONENT_NET_IFCFG`	interface configuration, must be done after interfaces are created
`RUMP_COMPONENT_DEV`	device components
`RUMP_COMPONENT_DEV_AFTERMAINBUS`	device components run which after the device tree root (*mainbus*) has attached
`RUMP_COMPONENT_KERN_VFS`	base initialization which is done after the VFS faction has attached, e.g. base components which do VFS operations
`RUMP_COMPONENT_SYSCALL`	establish *non-modular* syscalls which are available in the factions present
`RUMP_COMPONENT_POSTINIT`	misc. components that attach right before `rump_init()` returns

Table 3.4: **Component classes**. The `RUMP_COMPONENT` facility allows to specify component initialization at rump kernel bootstrap time. Due to interdependencies between subsystems, the component type specifies the order in which components are initialized. The order of component initialization is from top to bottom. The initializers designated `DEV`, `NET` and `VFS` are run only if the respective faction is present.

www.dbooks.org

```
RUMP_COMPONENT(RUMP_COMPONENT_NET)
{
        DOMAINADD(inetdomain);

        [ omitted: attach other domains ]
}

RUMP_COMPONENT(RUMP_COMPONENT_NET_IFCFG)
{
        [ omitted: local variables ]

        if ((error = socreate(AF_INET, &so, SOCK_DGRAM, 0, curlwp, NULL)) != 0)
                panic("lo0 config: cannot create socket");

        /* configure 127.0.0.1 for lo0 */
        memset(&ia, 0, sizeof(ia));
        strcpy(ia.ifra_name, "lo0");
        sin = (struct sockaddr_in *)&ia.ifra_addr;
        sin->sin_family = AF_INET;
        sin->sin_len = sizeof(struct sockaddr_in);
        sin->sin_addr.s_addr = inet_addr("127.0.0.1");

        [ omitted: define lo0 netmask and broadcast address ]

        in_control(so, SIOCAIFADDR, &ia, lo0ifp, curlwp);
        soclose(so);
}
```

Figure 3.17: **Example: selected contents of `netinet_component.c`.** The inet domain is attached in one constructor. The presence of the domain is required for configuring an inet address for `lo0`. The interface itself is provided and created by the *net* component (not shown).

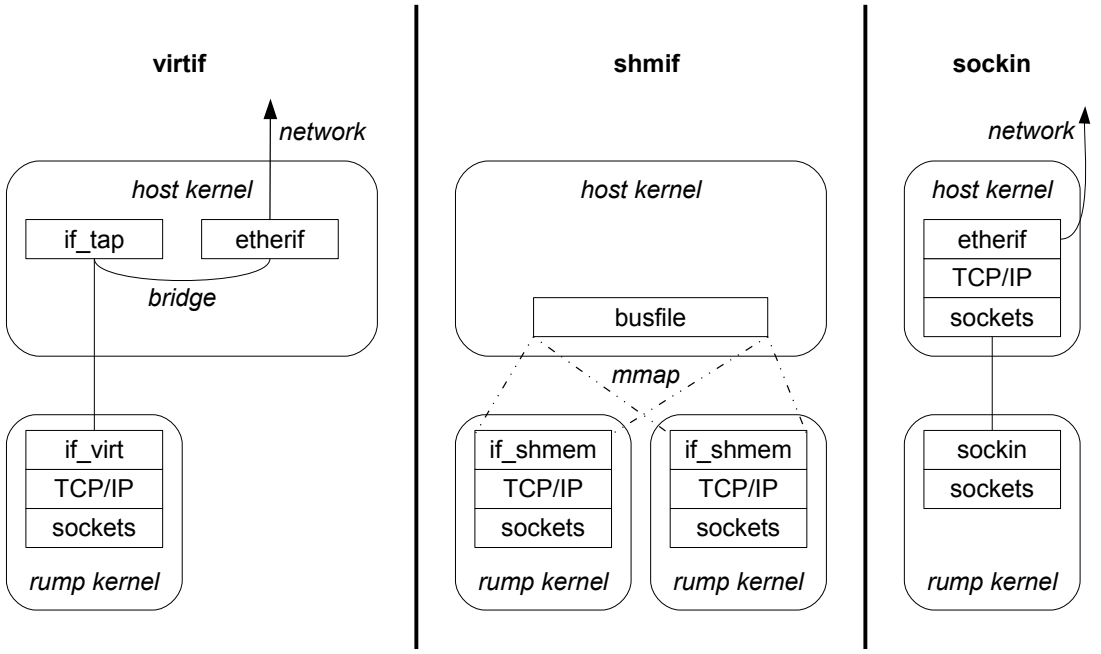


Figure 3.18: **Networking options for rump kernels.** The *virtif* facility provides a full networking stack by interfacing with, for example, the host's *tap* driver (depicted). The *shmem* facility uses interprocess shared memory to provide an Ethernet-like bus to communicate between multiple rump kernels on a single host without requiring elevated privileges on the host. The *sockin* facility provides unprivileged network access for all in-kernel socket users via the host's sockets.

3.9.1 Networking

The canonical way an operating system accesses the network is via an interface driver. The driver is at bottom of the network stack and has the capability for sending and receiving raw networking packets. On most general purpose OSs, sending and receiving raw network data is regarded as a privileged operation. Rather, unprivileged programs have only the capability to send and receive data via the sockets interfaces instead of deciding the full contents of a networking packet.

We have three distinct cases we wish to support in rump kernels. The cases are illustrated in Figure 3.18 and discussed next.

www.dbooks.org

1. **full network stack with raw access to the host's network**. In this case the rump kernel can send and receive raw network packets. An example of when this type of access is desired is an IP router. Elevated privileges are required, as well as selecting a host device through which the network is accessed.

2. **full network stack without access to the host's network**. In this use case we are interested in being able to send raw networking packets between rump kernels, but are not interested in being able to access the network on the host. Automated kernel driver testing in userspace is the main use case for this type of setup. We can fully use all of the networking stack layers, with the exception of the physical device driver, on a fully unprivileged account without any prior host resource allocation.

3. **unprivileged use of host's network for sending and receiving data**. In this case we are interested in the ability to send and receive data via the host's network, but do not care about the IP address associated with our rump kernel. Though sending data via the host's network stack requires that the host has a properly configured network stack, we argue that this assumption is commonly true.

   An example use case is the NFS client driver: we wish to isolate and virtualize the handling of the NFS protocol (i.e. the file system portion). However, we still need to transmit the data to the server. Using a full networking stack would not only require privileges, but also require configuring the networking stack (IP address, etc.). Using the host's stack to send and receive data avoids these complications.

   This option is directed at client side services, since all rump kernel instances will share the host's port namespace, and therefore it is not possible to start multiple instances of the same service.

```
# ifconfig tap0 create
# ifconfig tap0 up
# ifconfig bridge0 create
# brconfig bridge0 add tap0 add re0
# brconfig bridge0 up
```

Figure 3.19: **Bridging a tap interface to the host's `re0`.** The allows the tap device to send and receive network packets via `re0`.

Raw network access

The commonly available way for accessing an Ethernet network from a virtualized TCP/IP stack running in userspace is to use the *tap* driver with the `/dev/tap` device node. The tap driver presents a file type interface to packet networking, and raw Ethernet frames can be received and sent from an open device node using the `read()` and `write()` calls. If the tap interface on the host is bridged with a hardware Ethernet interface, access to a physical network is available since the hardware interface's traffic will be available via the tap interface as well. This tap/bridge scenario was illustrated in Figure 3.18.

The commands for bridging a tap interface on NetBSD are provided in Figure 3.19. Note that IP addresses are not configured for the tap interface on the host.

The *virt* (manual page *virt.4*) network interface driver we implemented uses hypercalls to open the tap device on the host and to transmit packets via it. The source code for the driver is located in `sys/rump/net/lib/libvirtif`.

Notably, while the tap method is near-ubiquitous and virtualizes the underlying network device for multiple consumers, it is not a high-performance option. At `http://repo.rumpkernel.org/` we host high-performance network driver implementations for DPDK and netmap [50] backends. While we will not discuss those

www.dbooks.org

implementations in this book, we want to note that the rump kernel side of the driver is shared between the one use for tap (i.e. *virt*). The difference comes from alternative hypercall implementations.

Full network stack without host network access

Essentially, we need an unprivileged Ethernet-like bus. All interfaces which are attached to the same bus will be able to talk to each other directly, while nodes with interfaces on other buses may be reached via routers.

One option for implementing packet distribution is to use a userland daemon which listens on a local domain socket [16]. The client kernels use hypercalls to access the local domain socket of the daemon. The daemon takes care of passing Ethernet frames to the appropriate listeners. The downside of the daemon is that there is an extra program to install, start and stop. Extra management is in conflict with the goals of rump kernels, and that is why we chose another implementation strategy.

We use shared memory provided by a memory mapped file as the network bus. The filename is the bus handle — all network interfaces on the same bus use the same filename. The *shmif* driver (manual page *shmif.4*) in the rump kernel accesses the bus. Each driver instance accesses one bus, so it is possible to connect a rump kernel to multiple different busses by configuring multiple drivers. Since all nodes have full access to bus contents, the approach does not protect against malicious nodes. As our main use case is testing, this lack of protection is not an issue. Also, the creation of a file on the host is not an issue, since testing is commonly carried out in a working directory which is removed after a test case has finished executing.

The shmif driver memory maps the file and uses it as a ring buffer. The header contains pointers to the first and last packet and a generation number, along with

bus locking information. The bus lock is a spinlock based on cross-process shared memory. A downside to this approach is that if a rump kernel crashes while holding the bus lock, the whole bus will halt. Since the main purpose of *shmif* is testing, we do not consider the bus halting a serious flaw.

Sending a packet requires locking the bus and copying the contents of the packet to the buffer. Receiving requires knowing when to check for new packets. It is the job of a hypercall to monitor the bus and wake up the receive side when the bus changes. This monitoring is, for example, implemented by using the *kqueue* facility on BSD and *inotify* on Linux. The receive side of the driver locks the bus and analyzes the bus header. If there are new packets for the interface on question, the driver passes them up to the IP layer.

An additional benefit of using a file is that there is always one ringbuffer's worth of traffic available in a postmortem situation. The `shmif_dumpbus` tool (manual page *shmif_dumpbus.1*) can be used to convert a busfile into the *pcap* format which the `tcpdump` tool understands. This conversion allows running a post-mortem tcpdump on a rump kernel's network packet trace.

Unprivileged use of the host's network in userspace

Some POSIX-hosted virtualization solutions such as QEMU and UML provide unprivileged zero-configuration network access via a facility called Slirp [1]. Slirp is a program which was popular during the dial-up era. It enables running a SLIP [52] endpoint on top of a regular UNIX shell without a dedicated IP. Slirp works by mapping the SLIP protocol to socket API calls. For example, when an application on the client side makes a connection, Slirp processes the SLIP frame from the client and notices a TCP SYN is being sent. Slirp then opens a socket and initiates a TCP connection on it. Note that both creating a socket and opening the connection are

www.dbooks.org

performed at this stage. This bundling happens because opening the socket in the application does not cause network traffic to be sent, and therefore Slirp is unaware of it.

Since the host side relies only on the socket API, there is no need to perform network setup on the host. Furthermore, elevated privileges are not required for the use of TCP and UDP (except for opening ports <1024). On the other hand, ICMP is not available since using it requires access to raw sockets on the host. Also, any IP address configured on the guest will be purely fictional, since socket traffic sent from Slirp will use the IP address of the host Slirp is running on.

Since Slirp acts as the peer for the guest's TCP/IP stack, it requires a complete TCP/IP stack implementation. The code required for complete TCP/IP processing is sizeable: over 10,000 lines of code.

Also, extra processing power is required, since the traffic needs to be converted multiple times: the guest converts the application data to IP datagrams, Slirp converts it back into data and socket family system call parameters, and the host's TCP/IP stack converts input from Slirp again to IP datagrams.

Our implementation is different from the one described above. Instead of doing transport and network layer processing in the rump kernel, we observe that regardless of what the guest does, processing will be done by the host. At best, we would need to undo what the guest did so that we can feed the payload data to the host's sockets interface. Instead of using the TCP/IP protocol suite in the rump kernel, we redefine the inet domain, and attach our implementation at the *protocol switch* layer [58]. We call this new implementation *sockin* to reflect it being **sock**et **in**et. The attachment to the kernel is illustrated in Figure 3.20. Attaching at the domain level means communication from the kernel's socket layer is done with *usrreq*'s, which in turn map to the host socket API in a very straightforward manner.

```
DOMAIN_DEFINE(sockindomain);

const struct protosw sockinsw[] = {
{
    .pr_type = SOCK_DGRAM, /* UDP */
    .pr_domain = &sockindomain,
    .pr_protocol = IPPROTO_UDP,
    .pr_flags = PR_ATOMIC | PR_ADDR,
    .pr_usrreq = sockin_usrreq,
    .pr_ctloutput = sockin_ctloutput,
},{
    .pr_type = SOCK_STREAM, /* TCP */
    .pr_domain = &sockindomain,
    .pr_protocol = IPPROTO_TCP,
    .pr_flags = PR_CONNREQUIRED | PR_WANTRCVD | PR_LISTEN | PR_ABRTACPTDIS,
    .pr_usrreq = sockin_usrreq,
    .pr_ctloutput = sockin_ctloutput,
}};

struct domain sockindomain = {
    .dom_family = PF_INET,
    .dom_name = "socket_inet",
    .dom_init = sockin_init,
    .dom_externalize = NULL,
    .dom_dispose = NULL,
    .dom_protosw = sockinsw,
    .dom_protoswNPROTOSW = &sockinsw[__arraycount(sockinsw)],
    .dom_rtattach = rn_inithead,
    .dom_rtoffset = 32,
    .dom_maxrtkey = sizeof(struct sockaddr_in),
    .dom_ifattach = NULL,
    .dom_ifdetach = NULL,
    .dom_ifqueues = { NULL },
    .dom_link = { NULL },
    .dom_mowner = MOWNER_INIT("",""),
    .dom_rtcache = { NULL },
    .dom_sockaddr_cmp = NULL
};
```

Figure 3.20: ***sockin* attachment.** Networking domains in NetBSD are attached by specifying a `struct domain`. Notably, the sockin family attaches a `PF_INET` type family since it aims to provide an alternative implementation for inet sockets.

www.dbooks.org

For example, for `PRU_ATTACH` we call `socket()`, for `PRU_BIND` we call `bind()`, for `PRU_CONNECT` we call `connect()`, and so forth. The whole implementation is 500 lines of code (including whitespace and comments), making it 1/20th of the size of Slirp.

Since sockin attaches as the Internet domain, it is mutually exclusive with the regular TCP/IP protocol suite. Furthermore, since the interface layer is excluded, the sockin approach is not suitable for scenarios which require full TCP/IP processing within the virtual kernel, e.g. debugging the TCP/IP stack. In such cases one of the other two networking models should be used. This choice may be made individually for each rump kernel instance.

3.9.2 Disk Driver

A disk block device driver provides storage medium access and is instrumental to the operation of disk-based file systems. The main interface is simple: a request instructs the driver to read or write a given number of sectors at a given offset. The disk driver queues the request and returns. The request is handled in an order according to a set policy, e.g. the disk head elevator. Once the request is complete, the driver signals the kernel that the request has been completed. In case the caller waits for the request to complete, the request is said to be synchronous, otherwise asynchronous.

There are two ways to provide a disk backend: buffered and unbuffered. A buffered backend stores writes to a buffer and flushes them to the backing storage later. Notably, to maintain file system on-disk correctness, synchronous writes must still be flushed to storage immediately. An unbuffered backend will always write to storage immediately. Examples of these backend types are a regular file and a character special device, respectively.

There are three approaches to implementing the block driver hypercalls using standard userspace interfaces.

- **Use `read()` and `write()` in caller context**: this is the simplest method. However, this method effectively makes all requests synchronous. Additionally, this method blocks other read operations when read-ahead is being performed.

- **Asynchronous read/write**: in this model the request is handed off to an I/O thread. When the request has been completed, the I/O thread signals completion.

  A buffered backend must flush synchronously executed writes. The only standard interface available for flushing is `fsync()`. However, it will flush all buffered data before returning, including previous asynchronous writes. Non-standard ranged interfaces such as `fsync_range()` exist, but they usually flush at least some file metadata in addition the actual data causing extra unnecessary I/O.

  A userlevel write to an unbuffered backend goes directly to storage. The system call will return only after the write has been completed. No flushing is required, but since userlevel I/O is serialized on Unix, it is not possible to issue another write before the first one finishes. This ordering means that a synchronous write must block and wait until any earlier write calls have been fully executed.

  The `O_DIRECT` file descriptor flag causes a write on a buffered backend to bypass cache and go directly to storage. The use of the flag also invalidates the cache for the written range, so it is safe to use in conjunction with buffered I/O. However, the flag is advisory. If conditions are not met, the I/O will silently fall back to the buffer. The direct I/O method can therefore be used only when it is certain that direct I/O applies.

www.dbooks.org

- **Memory-mapped I/O**: this method works only for regular files. The benefits are that the medium access fastpath does not involve any system calls and that the `msync()` system call can be portably used to flush ranges instead of the whole memory cache.

  The file can be mapped using windows. Windows provide two advantages. First, files larger than the available VAS can be accessed. Second, in case of a crash, the core dump is only increased by the size of the windows instead of the size of the entire file. We found that the number of windows does not have a significant performance impact; we default to 16 1MB windows with LRU recycling.

  The downside of the memory mapping approach is that to overwrite data, the contents must first be paged in, then modified, and only after that written. The pagein step is to be contrasted to explicit I/O requests, where it is possible to decide if a whole page is being written, and if so, skip pagein before write.

Of the above, we found that on buffered backends `O_DIRECT` works best. Ranged syncing and memory mapped I/O have roughly equal performance and full syncing performs poorly.

3.10 Hardware Devices: A Case of USB

A general purpose OS kernel USB driver stack may choose to export USB device access to userspace via the USB generic driver, or *ugen*. After ugen attaches to a USB bus node, it provides access to the attached hardware (i.e. not the entire USB bus) from userspace via the `/dev/ugen<n>` device nodes and read/write/ioctl calls. Providing access via a device node means that any entity on the host with the appropriate privileges to access the device node may communicate with the

hardware without having full access to the device registers. A key point is that the USB protocol offered by ugen is essentially unchanged from the USB hardware protocol. This protocol compatibility allows preexisting kernel drivers to use ugen without protocol translation.

At the root of the USB bus topology is a *USB host controller*. It controls all traffic on the USB bus. All device access on the bus is done through the host controller using an interface called USBDI, or USB Driver Interface. The role of the host controller, along with ugen, is a detail which makes USB especially suitable for userspace drivers: we need to implement a host controller which maps USBDI to the ugen device node instead of having to care about all bus details.

We implemented a host controller called *ugenhc*. When the kernel's device autoconfiguration subsystem calls the *ugenhc* driver to probe the device, the *ugenhc* driver tries to open `/dev/ugen` on the host. If the open is successful, the host kernel has attached a device to the respective ugen instance and *ugenhc* can return a successful match. Next, the *ugenhc* driver is attached in the rump kernel, along with a USB bus and a USB root hub. The root hub driver explores the bus to see which devices are connected to it, causing the probes to be delivered first to *ugenhc* and through `/dev/ugen` to the host kernel and finally to the actual hardware. The device driver instance can be used from the rump kernel just like any other device driver, e.g. USB network interfaces can be used by the networking stack to shuffle packets.

3.10.1 Conclusions of the USB Approach

The USB approach is not recommended for hooking device drivers to rump kernels. To implement *ugenhc*, support for the USB protocol stack must already exist on the host. Even though USB is supported by many hosts, the current implementation of *ugenhc* is still NetBSD-specific. Furthermore, we found that though in theory

www.dbooks.org

access through `/dev/ugen` is safe, during development we were able to tickle the host's USB protocol stack in surprising ways causing host kernel panics. Supposedly, this instability is caused by the compound effect of both the USB protocol stack implementation being huge, and accessing it via `/dev/ugen` not being the frequently exercised access path.

3.11 Microkernel Servers: Case Study with File Servers

In this section we investigate using rump kernels as microkernel style servers for file systems. Our key motivation is to prevent a malfunctioning file system driver from damaging the host kernel by isolating it in a userspace server.

The NetBSD framework for implementing file servers in userspace is called *puffs* [28]. We use puffs to attach the rump kernel file server to the host's file system namespace. Conceptually, after the file system has been mounted, the service works as follows: a file system request is transported from the host kernel to the userspace server using puffs. The server makes a local call into the rump kernel to service the request. When servicing the request is complete, the response is returned to the host kernel using puffs. The architecture of this solution is presented in Figure 3.21. It is worth noting that a userlevel application is not the only possible consumer. Any VFS user, such as an NFS server running in the host kernel, is a valid consumer in this model.

3.11.1 Mount Utilities and File Servers

Before a file system can be accessed, it must be mounted. Standard kernel file systems are mounted with utilities such as `mount_efs`, `mount_tmpfs`, etc. These utilities parse the command line arguments and call the `mount()` system call with a file system specific argument structure built from the command line arguments.

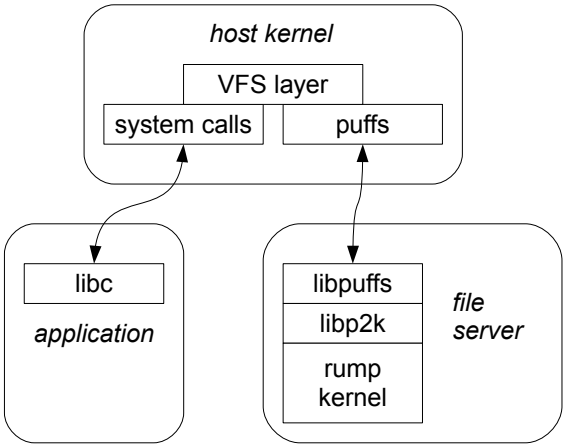


Figure 3.21: **File system server.** The request from the microkernel client is transported by the host kernel to the rump kernel running providing the kernel file system driver. Although only system calls are illustrated, page faults created by the client may be handled by the server as well.

One typical way of invoking these utilities is to use the `mount` command with an argument specifying the file system. For example, `mount -t efs /dev/sd0e /mnt` invokes `mount_efs` to do the actual mounting.

Instead of directly calling `mount()`, our server does the following: we bootstrap a rump kernel, mount the file system in the rump kernel, and attach this process as a puffs server to the host. All of these tasks are performed by our mount commands counterparts: `rump_efs`, `rump_tmpfs`, etc. The usage of the rump kernel variants is unchanged from the originals, only the name is different. To maximize integration, these file servers share the same command line argument parsing code with the regular mount utilities. Sharing was accomplished by restructuring the mount utilities to provide an interface for command line argument parsing and by calling those interfaces from the `rump_xfs` utilities.

Sharing argument parsing means that the file servers have the same syntax. This feature makes usage interchangeable just by altering the command name. We also

www.dbooks.org

in-kernel mount:

```
/dev/sd0e                10.181.181.181:/m/dm    /m/usb    /m/dm    msdos    nfs    rw,-u=1000    rw,-p
```

equivalent rump kernel file server mount:

```
/dev/sd0e                10.181.181.181:/m/dm    /m/usb    /m/dm    msdos    nfs    rw,-u=1000,rump    rw,-p,rump
```

Figure 3.22: **Use of `-o rump` in `/etc/fstab`.** The syntax for a file system served by an in-kernel driver or a rump kernel is the same apart from the *rump* flag.

added a `rump` option to the `mount` command. For example, consider the following command: `mount -t efs -o rump /dev/sd0e /mnt`. It will invoke `rump_efs` instead of `mount_efs` and therefore the file system will be mounted with a rump kernel file system driver. The `rump` option works also in `/etc/fstab`, as is illustrated in Figure 3.22. The flag allows the use of rump kernel file servers to handle specific mounts such as USB devices and CD/DVD by adding just one option. The figure also demonstrates how the NFS client (same applies to SMBFS/CIFS) running inside a rump kernel or the host kernel are completely interchangeable since the rump kernel drivers use the sockin networking facility (Section 3.9.1) and therefore share the same IP address with the host.

The list of kernel file system drivers available as rump servers is available in the "SEE ALSO" section of the `mount(8)` manual page on a NetBSD system. Support in 5.99.48 consists of ten disk-based and two network-based file systems.

3.11.2 Requests: The p2k Library

We attach to the host as a puffs file server, so the file system requests we receive are in the format specified by puffs. We must feed the requests to the rump kernel to access the backend file system. To be able to do so, we must convert the requests to a suitable format. Since the interface offered by puffs is close to the kernel's VFS/vnode interface [36] we can access the rump kernel directly at the VFS/vnode layer if we translate the puffs protocol to the VFS/vnode protocol.

We list some examples of differences between the puffs protocol and VFS/vnode protocol that we must deal with by translations. For instance, the kernel references a file using a `struct vnode` pointer, whereas puffs references one using a `puffs_cookie_t` value. Another example of a difference is the way $(address, size)$-tuples are indicated. In the kernel `struct uio` is used. In puffs, the same information is passed as separate pointer and byte count parameters.

The p2k, or puffs-to-kernel, library is a request translator between the puffs userspace file system interface and the kernel virtual file system interface (manual page *p2k.3*). It also interprets the results from the kernel file systems and converts them back to a format that puffs understands.

Most of the translation done by the p2k library is a matter of converting data types back and forth. To give an example of p2k operation, we discuss reading a file, which is illustrated by the p2k read routine in Figure 3.23. We see the uio structure being created by `rump_uio_setup()` before calling the vnode operation and being freed after the call while saving the results. We also notice the puffs credit type being converted to the opaque `kauth_cred_t` type used in the kernel. This conversion is done by the p2k library's `cred_create()` routine, which in turn uses `rump_pub_cred_create()`.

www.dbooks.org

```
int
p2k_node_read(struct puffs_usermount *pu, puffs_cookie_t opc,
        uint8_t *buf, off_t offset, size_t *resid, const struct puffs_cred *pcr, int ioflag)
{
        struct vnode *vp = OPC2VP(opc);
        struct kauth_cred *cred = cred_create(pcr);
        struct uio *uio = rump_pub_uio_setup(buf, *resid, offset, RUMPUIO_READ);
        int rv;

        RUMP_VOP_LOCK(vp, LK_SHARED);
        rv = RUMP_VOP_READ(vp, uio, ioflag, cred);
        RUMP_VOP_UNLOCK(vp);
        *resid = rump_pub_uio_free(uio);
        cred_destroy(cred);

        return rv;
}
```

Figure 3.23: **Implementation of `p2k_node_read()`.** The parameters from the puffs interface are translated to parameters expected by the kernel vnode interface. Kernel data types are not exposed to userspace, so rump kernel public routines are used to allocate, initialize and release such types.

The `RUMP_VOP_LOCK()` and `RUMP_VOP_UNLOCK()` macros deal with NetBSD kernel VFS locking protocol. They take a lock on the vnode and unlock it, respectively. From one perspective, locking at this level is irrelevant, since puffs in the host kernel takes care of locking. However, omitting lock operations from the rump kernel causes assertions such as `KASSERT(VOP_ISLOCKED(vp));` in kernel drivers to fire. Therefore, proper locking is necessary at this layer to satisfy the driver code.

3.11.3 Unmounting

A p2k file server can be unmounted from the host's namespace in two ways: either using the `umount` command (and the `unmount()` system call) on the host or by killing the file server. The prior method is preferred, since it gives the kernel cache

```
golem> mount -t msdos -o rump /dev/sd0e /mnt
panic: buf mem pool index 23
Abort (core dumped)
golem>
```

Figure 3.24: **Mounting a corrupt FAT FS with the kernel driver in a rump kernel.** If the file system would have been mounted with the driver running in the host kernel, the entire host would have crashed. With the driver running in userspace in a rump kernel, the mount failed and a core dump was created without otherwise affecting the host.

in puffs a chance to flush all data. It also allows the p2k library to call the rump kernel and ask it to unmount the file system and mark it clean.

3.11.4 Security Benefits

Drivers for disk-based file systems are written assuming that file system images contain trusted input. With USB sticks and DVDs untrusted images are common. Still, without thinking, users mount untrusted file systems using kernel drivers. Arbitrary memory access is known to be possible using a suitable crafted file system image and fixing each file system driver to be bullet-proof is at best difficult [59].

When run in a rump kernel, a file system driver dealing with an untrusted image is isolated in its own domain. This separation mitigates the possibility of a direct memory access attack on the kernel.

To give an example of a useful scenario, a mailing list posting described a problem with mounting a FAT file system from a USB stick causing a kernel crash and complete system failure. By using a rump kernel with microkernel clients, the problem is only an application core dump. Figure 3.24 illustrates what happens when the file

www.dbooks.org

system is mounted with the driver running in a rump kernel. Of course, the driver was fixed to deal graciously with this particular bug, but others remain.

It needs to be stressed that mounting a file system as a server is feature wise no different than using a driver running in the host kernel. The user and administrator experience remains the same, and so does the functionality. Only the extra layer of security is added. It is the author's opinion and recommendation that untrusted disk file systems should be never be mounted using a file system driver running in kernel space.

A rump kernel has the same privileges as a process, so from the perspective of the host system its compromise is the same as the compromise of any other application. In case rogue applications are a concern, on most operating systems access can be further limited by facilities such as jails [27] or sandboxing [19]. Networked file system clients (such as NFS and CIFS) may also benefit from the application of firewalls.

3.12 Remote Clients

Remote clients are clients which are disjoint from the rump kernel. For example, on POSIX hosts remote clients run in different processes than their respective rump kernels, either on the same host or not. The advantage of a remote client is that the relationship between the remote client and a rump kernel is much like that of a regular kernel and a process: the clients start up, run and exit independently. This independence makes it straightforward to adapt existing programs as rump kernel clients, and as we will see later in this section, allows existing POSIX binaries to use services from a rump kernel without recompilation. For example, it is possible to hijack an unmodified Firefox browser to use a TCP/IP stack provided by a rump kernel. Remote clients are also the basis for rumpctrl (Section 4.3).

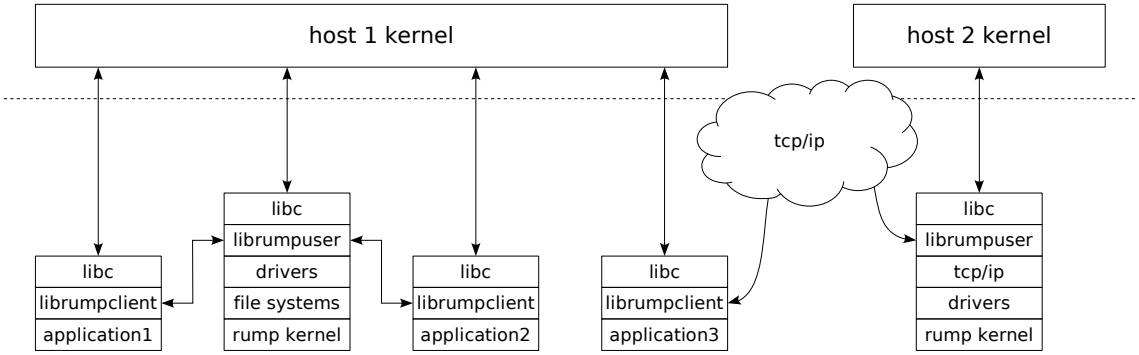


Figure 3.25: **Remote client architecture.** Remote clients communicate with the rump kernel through the *rumpclient* library. The client and rump kernel may or may not reside on the same host or same type of host.

The general architecture of remote rump clients is illustrated in Figure 3.25. It is explained in detail in the following sections.

Communication can be done over virtually any type of bus. The requirements for the bus are merely to be able to read and write datagrams. By default, we provide the implementation for two socket-based protocol families: Unix domain sockets and TCP/IP. The advantages of Unix domain sockets are that the available namespace is virtually unlimited and it is easy to bind a private server in a local directory without fear of a resource conflict. Also, it is possible to use host credentials (via `chmod`) to control who has access to the server. The TCP method does not have these advantages — in the general case it is not possible to guarantee that a predefined port is not in use — but TCP does work over the Internet. Generally speaking, Unix domain sockets should be used when the server and client reside on the same host.

www.dbooks.org

3.12.1 Client-Kernel Locators

Before the client is able to contact the rump kernel, the client must know where the kernel is located. In the traditional Unix model locating the kernel is simple, since there is one unambiguous kernel ("host kernel") which is the same for every process. However, remote clients can communicate with any rump kernel which may or may not reside on the same host.

The client and rump kernel find each other by specifying a location using a URL. For example, the URL `tcp://1.2.3.4:4321/` specifies a TCP connection on IP address 1.2.3.4 port 4321, while `unix://serversocket` specifies a UNIX domain socket *relative* to the current working directory.

While service discovery models [11] are possible, they are beyond our current scope, and manual configuration is currently required. In most cases, such as for all the rump kernel using tests we have written, the URL can simply be hardcoded.

3.12.2 The Client

A remote client, unlike a local client, is not linked against the rump kernel. Instead, it is linked against *librumpclient* (manual page *rumpclient.3*). Linking can happen either when a program is compiled or when it is run — we mentioned this aspect of linking earlier in Section 3.1.2. The former approach is usually used when writing programs with explicit rump kernel knowledge. The latter approach uses the dynamic linker and can be used for pre-existing programs which were written and compiled without knowledge of a rump kernel.

The rumpclient library provides support for connecting to a rump kernel and abstracts communication between the client and the server. Furthermore, it provides

function interfaces for system calls, as described in Section 3.6.1. Other interfaces such as the VFS interfaces and rump kernel private interfaces are *not* provided, since they do not implement the appropriate access control checks for remote clients.

The librumpclient library supports both singlethreaded and multithreaded clients. Multithreaded clients are supported transparently, i.e. all the necessary synchronization is handled internally by the library. The librumpclient library also supports persistent operation, meaning it can be configured to automatically try to reconnect in case the connection with the server is severed. Notably, as a reconnect may mean for instance that the kernel server crashed and was restarted, the applications using this facility need to be resilient against kernel state loss. One example is a web browser, which requires only a page reload in case a TCP/IP server was killed in the middle of loading a page.

The server URL is read from the `RUMP_SERVER` environment variable. The environment is used instead of a command line parameter so that applications which were not originally written to be rump kernel clients can still be used as rump kernel clients without code changes.

3.12.3 The Server

A rump kernel can be configured as a server by calling the rump kernel interface `rump_init_server(const char *url)` from the local client. The argument is a URL indicating an address the server will be listening to. The server will handle remote requests automatically in the background. Initializing the serverside will not affect the local client's ability to communicate with the rump kernel.

The `rump_server` daemon (manual page *rump_server.1*) is a configurable userspace daemon for serving rump kernel remote clients. The factions and drivers supported

www.dbooks.org

A tmpfs server listening on `INADDR_ANY` port 12765:

```
$ rump_server -lrumpvfs -lrumpfs_tmpfs tcp://0:12765/
```

Map 1GB host file `dk.img` as the block device `/dev/dk` using etfs, specify local domain URL using a relative path:

```
$ rump_allserver -d key=/dev/dk,hostpath=dk.img,size=1g unix://dkserv
```

A TCP/IP server with the `if_virt` driver, specify socket using an absolute path:

```
$ rump_server -lrumpnet -lrumpnet_net -lrumpnet_netinet \
    -lrumpnet_virt unix:///tmp/tcpip
```

Figure 3.26: **Example invocations for `rump_server`.** All invocations create rump kernels listening for clients at different addresses with different capabilities.

by the server instance are given as command line arguments and dynamically loaded by the server. The variant `rump_allserver` includes all rump kernel components that were available at the time that the system was built. Figure 3.26 illustrates server usage with examples.

The data transport and protocol layer for remote clients is implemented entirely within the hypervisor. The original implementation [9] utilized userspace host sockets, so the hypervisor seemed like a convenient place to implement support. Later, that also turned out to be the only sensible place, as it avoids having to teach the rump kernel about the specifics of each bus.

The implementation locus means that the kernel side of the server and the hypercall layer need to communicate with each other. The interfaces used for communication are a straightforward extension of the protocol we will discuss in detail next

[9] Actually, that was the second implementation, but it was the first implementation which was not purely experimental.

(Section 3.12.4); we will not discuss the interfaces. The interfaces are defined in `sys/rump/include/rump/rumpuser.h` and are implemented for the POSIX hypervisor in `lib/librumpuser/rumpuser_sp.c`. The rump kernel side of the implementation is provided by the `rumpkern_sysproxy` component. As a corollary of the implementation being a component, support is optional in any given rump kernel instance.

3.12.4 Communication Protocol

The communication protocol between the client and server is a protocol where the main feature is a system call. The rest of the requests are essentially support features for the system call. To better understand the request types, let us first look at an example of what happens when a NetBSD process requests the opening of `/dev/null` from a regular kernel.

1. The user process calls the routine `open("/dev/null", O_RDWR);`. This routine resolves to the system call stub in libc.

2. The libc system call stub performs a system call trap causing a context switch to the kernel. The calling userspace thread is suspended until the system call returns.

3. The kernel receives the request, examines the arguments and determines which system call the request was for. It begins to service the system call.

4. The path of the file to be opened is required by the kernel. A pointer to the path string is passed as part of the arguments. The string is copied in from the process address space only if it is required. The `copyinstr()` routine is called to copy the pathname from the user process address space to the kernel address space.

www.dbooks.org

5. The file system code does a lookup for the pathname. If the file is found, and the calling process has permissions to open it in the mode specified, and various other conditions are met, the kernel allocates a file descriptor for the current process and sets it to reference a file system node describing `/dev/null`.

6. The system call returns the fd (or error along with `errno`) to the user process and the user process continues execution.

We created a communication protocol between the client and rump kernel which supports interactions of the above type. The request types from the client to the kernel are presented and explained in Table 3.5 and the requests from the kernel to the client are presented and explained in Table 3.6.

Request	Arguments	Response	Description
handshake	type (guest, authenticated or exec), name of client program	success/fail	Establish or update a process context in the rump kernel.
syscall	syscall number, syscall args	return value, errno	Execute a system call.
prefork	none	authentication cookie	Establish a fork authentication cookie.

Table 3.5: **Requests from the client to the kernel.**

Now that we know the communication protocol, we will compare the operations executed in the regular case and in the rump kernel remote client case side-by-side. The first part of the comparison is in Table 3.7 and the second part is in Table 3.8.

Request	Arguments	Response	Description
copyin + copyinstr	client address space pointer, length	data	The client sends data from its address space to the kernel. The "str" variant copies up to the length of a null-terminated string, i.e. length only determines the maximum. The actual length is implicitly specified by the response frame length.
copyout + copyoutstr	address, data, data length	none (kernel does not expect a response)	Requests the client to copy the attached data to the given address in the client's address space.
anonmmap	mmap size	address anon memory was mapped at	Requests the client to mmap a window of anonymous memory. This request is used by drivers which allocate userspace memory before performing a copyout.
raise	signal number	none (kernel does not expect a response)	Deliver a host signal to the client process. This request is used to implement the rump kernel "raise" signal model.

Table 3.6: Requests from the kernel to the client.

www.dbooks.org

Host syscall	rump syscall
1. `open("/dev/null", O_RDWR)`	1. `rump_sys_open("/dev/null", O_RDWR)` is called
2. libc executes the syscall trap.	2. librumpclient marshals the arguments and sends a "syscall" request over the communication socket. the calling thread is suspended until the system call returns.
3. syscall trap handler calls `sys_open()`	3. rump kernel receives syscall request and uses a thread associated with the process to handle request 4. thread is scheduled, determines that `sys_open()` needs to be called, and proceeds to call it.
4. pathname lookup routine calls `copyinstr()`	5. pathname lookup routine needs the path string and calls `copyinstr()` which sends a copyinstr request to the client 6. client receives copyinstr request and responds with string datum 7. kernel server receives a response to its copyinstr request and copies the string datum to a local buffer

Table 3.7: Step-by-step comparison of host and rump kernel syscalls, part 1/2.

Host syscall	rump syscall
5. the lookup routine runs and allocates a file descriptor referencing a backing file system node for `/dev/null`	8. same
6. the system call returns the fd	9. the kernel sends the return values and errno to the client 10. the client receives the response to the syscall and unblocks the thread which executed this particular system call 11. the calling thread wakes up, sets errno (if necessary) and returns with the return value received from the kernel

Table 3.8: Step-by-step comparison of host and rump kernel syscalls, part 2/2.

www.dbooks.org

3.12.5 Of Processes and Inheritance

The process context for a remote client is controlled by the rump kernel server. The rump_lwproc interfaces available for local clients (manual page *rump_lwproc.3*) cannot be used by remote clients. Whenever a client connects to a rump kernel and performs a handshake, a new process context is created in the rump kernel. All requests executed through the same connection are executed on the same rump kernel process context.

A client's initial connection to a rump kernel is like a login: the client is given a rump kernel process context with the specified credentials. After the initial connection, the client builds its own process family tree. Whenever a client performs a fork after the initial connection, the child must inherit both the properties of the host process and the rump kernel process to ensure correct operation. When a client performs exec, the process context must not change.

Meanwhile, if another client, perhaps but not necessarily from another physical machine, connects to the rump kernel server, it gets its own pristine login process and starts building its own process family tree through forks.

By default, all new connections currently get root credentials by performing a *guest* handshake. We recognize that root credentials are not always optimal in all circumstances, and an alternative could be a system where cryptographic verification is used to determine the rump kernel credentials of a remote client. Possible examples include Kerberos [39] and TLS [47] with clientside certificates. For the use cases so far, limiting access to the server URL has been sufficient.

When a connection is severed, the meaning of which is bus-dependent, the rump kernel treats the process context as a killed process. The rump kernel wakes up any and all threads associated with the connection currently blocking inside the rump

kernel, waits for them to exit, and then proceeds to free all resources associated with the process.

3.12.6 Host System Call Hijacking

The only difference in calling convention between a rump client syscall function and the corresponding host syscall function in libc is the `rump_sys`-prefix for a rump kernel syscall. In other words, it is possible to select the entity the service is requested from by adding or removing a prefix from the system call name. The benefit of explicit source-level selection is that there is full control of which system call goes where. The downside is that it requires source level control and compilation. To use unmodified binaries, we must come up with a policy which determines which kernel handles each syscall.

A key point for us to observe is that in Unix a function call API in libc (e.g. `open(const char *path, int flags, mode_t mode)`) exists for all system calls. The libc stub abstracts the details of user-kernel communication. The abstraction makes it possible to change the nature of the call just by intercepting the call to `open()` and directing it elsewhere. If the details of making the request were embedded in the application itself, it would be much more difficult to override them to call a remote rump kernel instead of the local host kernel.

The rumphijack library (`lib/librumphijack`, Figure 3.27) provides a mechanism and a configurable policy for unmodified applications to capture and route part of their system calls to a rump kernel instead of the host kernel. Rumphijack is based on the technique of using `LD_PRELOAD` to instruct the dynamic linker to load a library so that all unresolved symbols are primarily resolved from that library. The library provides its own system call stubs that select which kernel the call should go to. While the level of granularity is not per-call like in the explicit source control

www.dbooks.org

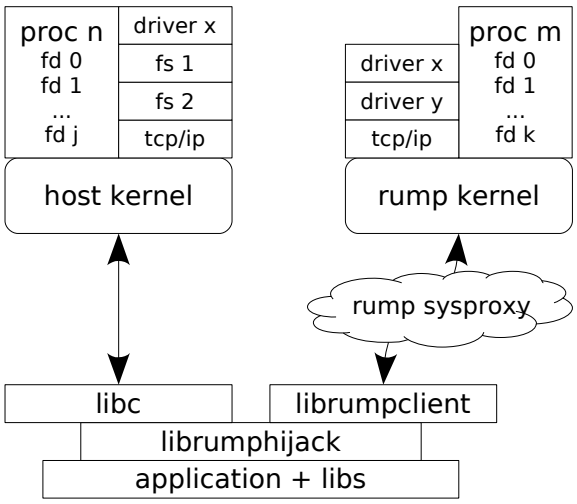


Figure 3.27: **System call hijacking.** The *rumphijack* library intercepts system calls and determines whether the syscall request should be sent to the rump kernel or the host kernel for processing.

method, using the classification technique we present below, this approach works in practice for all applications.

From the perspective of librumphijack, system calls can be divided into roughly the following categories. These categories determine where each individual system call is routed to.

- **purely host kernel calls**: These system calls are served only by the host kernel and never the rump kernel, but nevertheless require action on behalf of the rump kernel context. Examples include `fork()` and `execve()`.
- **create an object**: the system call creates a file descriptor. Examples include `open()` and `accept()`.
- **decide the kernel based on an object identifier**: the system call is

directed either to the host kernel or rump kernel based on a file descriptor value or pathname.

- **globally pre-directed to one kernel**: the selection of kernel is based on user configuration rather than parameter examination. For example, calls to `socket()` with the same parameters will always be directed to a predefined kernel, since there is no per-call information available.

- **require both kernels to be called simultaneously**: the asynchronous I/O calls (`select()`, `poll()` and variants) pass in a list of descriptors which may contain file descriptors from both kernels.

Note: the categories are not mutually exclusive. For example, `socket()` and `open()` belong to several of them. In case `open()` is given a filename under a configurable prefix (e.g. `/rump`), it will call the rump kernel to handle the request and new rump kernel file descriptor will be returned to the application as a result.

The rest of this section describes advanced rumphijack features beyond simple system call routing. Nevertheless, those features are commonly required for supporting many real-world applications.

File Descriptor Games

A rump kernel file descriptor is differentiated from a host kernel file descriptor by the numerical value of the file descriptor. Before a rump kernel descriptor is returned to the application, it is offset by a per-process configurable constant. Generally speaking, if the file descriptor parameter for a system call is greater than the offset, it belongs to the rump kernel and the system call should be directed to the rump kernel.

www.dbooks.org

The default offset was selected to be half of `select()`'s `FD_SETSIZE` and is 128. This value allows almost all applications to work, including historic ones that use `select()` and modern ones that use a fairly large number of file descriptors. In case the host returns a file descriptor which is equal to or greater than the process's hijack fd offset, rumphijack closes the fd and sets errno to `ENFILE`.

A problem arises from the `dup2()` interface which does not fit the above model: in dup2 the new file descriptor number is decided by the caller. For example, a common scheme used e.g. by certain web servers is accepting a connection on a socket, forking a handler, and dup2'ing the accepted socket connection to stdin/stdout. The new file descriptor must belong to the same kernel as the old descriptor, but in case of stdin/stdout, the new file descriptor numbers always signify the host kernel. To solve this conflict, we maintain a file descriptor aliasing table which keeps track of cross-kernel dup2's. There are a number of details involved, such as making sure that closing the original fd does not close the dup2'd fd in the different kernel namespace, and making sure we do not return a host descriptor with a value duplicate to one in the dup2 space. In fact, a large portion of the code in the hijack library exists solely to deal with complexities related to dup2. All of the complexity is fully contained within the hijack and rumpclient libraries and it is not visible to applications using the libraries.

Another issue we must address is protecting the file descriptors used internally by librumpclient. Recall, the connection between the remote client and the rump kernel associates the remote client with a rump kernel process context, and if the connection is lost all rump kernel process state such as file descriptors are lost with it. In some scenarios applications want to close file descriptors en masse. One example of such a scenario is when an application prepares to call `exec()`. There are two approaches to mass closing: either calling `close()` in a loop up to an arbitrary descriptor number or calling `closefrom()` (which essentially calls `fcntl(F_DUPFD)`). Since the application never sees the rumpclient internal file descriptors and hence should

not close them, we take precautions to prevent it from happening. The hijack library notifies rumpclient every time a descriptor is going to be closed. There are two distinct cases:

- A call closes an individual host descriptor. In addition to the obvious `close()` call, `dup2()` also belongs into this category. Here we inform rumpclient of a descriptor being closed and in case it is a rumpclient descriptor, it is dup'd to another value, after which the hijack library can proceed to invalidate the file descriptor by calling close or dup2.

- The `closefrom()` routine closes all file descriptors equal to or greater than the given descriptor number. We handle this operation in two stages. First, we loop and call `close()` for all descriptors which are not internal to rumpclient. After we reach the highest rumpclient internal descriptor we can execute a host `closefrom()` using one greater than the highest rumpclient descriptor as the argument. Next, we execute `closefrom()` for the rump kernel, but this time we avoid closing any dup2'd file descriptors.

Finally, we must deal with asynchronous I/O calls that may have to call both kernels. For example, in networking clients it is common to pass in one descriptor for the client's console and one descriptor for the network socket. Since we do not have a priori knowledge of which kernel will have activity first, we must query both. This simultaneous query is done by creating a thread to call the second kernel. Since only one kernel is likely to produce activity, we also add one host kernel pipe and one rump kernel pipe to the file descriptor sets being polled. After the operation returns from one kernel, we write to the pipe of the other kernel to signal the end of the operation, join the thread, collect the results, and return.

www.dbooks.org

3.12.7 A Tale of Two Syscalls: `fork()` and `execve()`

The `fork()` and `execve()` system calls require extra consideration both on the client side and the rump kernel side due to their special semantics. We must preserve those semantics both for the client application and the rump kernel context. While these operations are implemented in librumpclient, they are most relevant when running hijacked clients. Many programs such as the OpenSSH [44] `sshd` or the mutt [42] MUA fail to operate as remote rump clients if support is handled incorrectly.

Supporting `fork()`

Recall, the `fork()` system call creates a copy of the calling process which essentially differs only by the process ID number. After forking, the child process shares the parent's file descriptor table and therefore it shares the rumpclient socket. A shared connection cannot be used, since use of the same socket from multiple independent processes will result in corrupt transmissions. Another connection must be initiated by the child. However, as stated earlier, a new connection is treated like an initial login and means that the child will not have access to the parent's rump kernel state, including file descriptors. Applications such as web servers and shell input redirection depend on the behavior of file descriptors being correctly preserved over fork.

We solve the issue by dividing forking into three phases. First, the forking process informs the rump kernel that it is about to fork. The rump kernel does a fork of the rump process context, generates a cookie and sends that to the client as a response. Next, the client process calls the host's fork routine. The parent returns immediately to the caller. The newly created child establishes its own connection to the rump kernel server. It uses the cookie to perform a handshake where it indicates it wants to attach to the rump kernel process the parent forked off earlier. Only

```
pid_t
rumpclient_fork()
{
        pid_t rv;

        cookie = rumpclient_prefork();
        switch ((rv = host_fork())) {
        case 0:
                rumpclient_fork_init(cookie);
                break;
        default:
                break;
        case -1:
                error();
        }

        return rv;
}
```

Figure 3.28: **Implementation of `fork()` on the client side.** The prefork cookie is used to connect the newly created child to the parent when the new remote process performs the rump kernel handshake.

then does the child return to the caller. Both host and rump process contexts retain expected semantics over a host process fork. The client side `fork()` implementation is illustrated in Figure 3.28. A hijacked fork call is a simple case of calling `rumpclient_fork()`.

Supporting execve()

The requirements of exec are the "opposite" of fork. Instead of creating a new process, the same rump process context must be preserved over a host's exec call.

Since calling exec replaces the memory image of a process with that of a new one from disk, we lose all of the rump client state in memory. Important state in memory

www.dbooks.org

includes for example rumpclient's file descriptors. For hijacked clients the clearing of memory additionally means we will lose e.g. the dup2 file descriptor alias table. Recall, though, that exec closes only those file descriptors which are set `FD_CLOEXEC`.

Before calling the host's execve, we first augment the environment to contain all the rump client state; librumpclient and librumphijack have their own sets of state as was pointed out above. After that, `execve()` is called with the augmented environment. When the rump client constructor runs, it will search the environment for these variables. If found, it will initialize state from them instead of starting from a pristine state.

As with fork, most of the kernel work is done by the host system. However, there is also some rump kernel state we must attend to when exec is called. First, the process command name changes to whichever process was exec'd. Furthermore, although file descriptors are in general not closed during exec, ones marked with `FD_CLOEXEC` should be closed, and we call the appropriate kernel routine to have them closed.

The semantics of exec also require that only the calling thread is present after exec. While the host takes care of removing all threads from the client process, some of them might have been blocking in the rump kernel and will continue to block until their condition has been satisfied. If they alter the rump kernel state after their blocking completes at an arbitrary time in the future, incorrect operation may result. Therefore, during exec we signal all lwps belonging to the exec'ing process that they should exit immediately. We complete the exec handshake only after all such lwps have returned from the rump kernel.

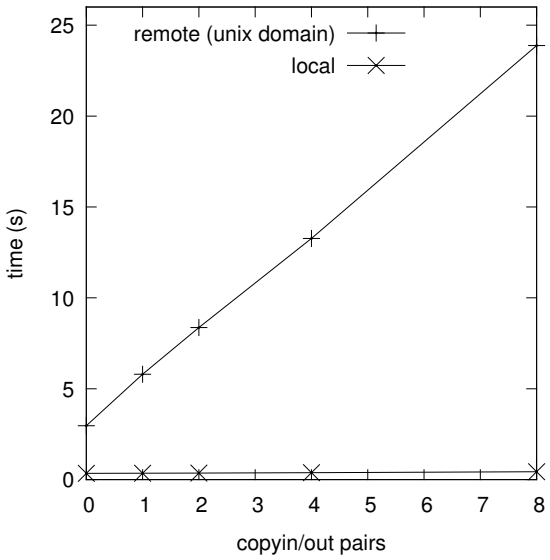


Figure 3.29: **Local vs. Remote system call overhead.** The cost of remote system calls is dominated by the amount of client-kernel roundtrips necessary due to copying data in and out. For local clients the cost of a system call is virtually independent of the amount of copies in and out.

3.12.8 Performance

Figure 3.29 shows the amount of time it takes to perform 100,000 system call requests as a function of the amount of copyin/out pairs required for servicing the system call. A system call which does nothing except copyin/out on 64 byte buffers was created for the experiment. The measurement was done both for a local client and a remote client accessing a server hosted on the same system. We see that for the remote client copyin/out dominates the cost — if the system call request itself is interpreted as a copyin and copyout operation, the time is a linear function of the number of copyin/out operations. In contrast, for the local case the duration increases from 0.34s to 0.43s when going from 0 to 8 copyin/out requests. This data shows that copyin/out I/O is a factor in total cost for local calls, but it does not have a dominant impact. Therefore, we conclude that the IPC between the client and server is the dominating cost for remote system calls.

www.dbooks.org

The straightforward optimization which does not involve modifying the host system is to decrease the number of remote copyin/out requests required for completing a syscall request. This decrease can be reached in a fairly straightforward manner by augmenting the syscall definitions and pre-arranging parameters so that pre-known copyin/out I/O can be avoided. Possible options are piggy-backing the data copy as part of syscall request/response, or by using interprocess shared memory in case the client and server are on the same machine. For example, the `open()` syscall will, barring an early error, always copy in the pathname string. We can make the syscall code set things up so that the pathname copyin is immediately satisfied with a local copy operation instead of a remote request and the associated round trip delay.

Anecdotal analysis

For several weeks the author did his day-to-day web browsing with Firefox acting as a remote client for a rump kernel TCP/IP stack. There was no human-perceivable difference between the performance of a rump networking stack and the host networking stack, either in bulk downloads, flash content or interactive page loads. The only human-perceivable difference was the ability to reboot the TCP/IP stack from under the browser without having to close the browser first.

Microbenchmarks show that remote system calls are orders of magnitude slower than local system calls especially due to copyin/out I/O. However, this "macrobenchmark" suggests that others factors in real application hugely mitigate this performance difference. We conclude that without a specific application use case any optimizations are premature. In the event of such use cases emerging, optimizations know from literature [6, 31] may be attempted.

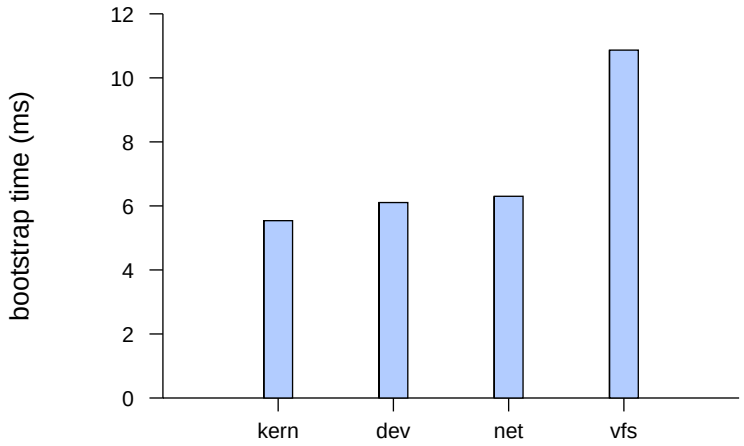


Figure 3.30: **Time required to bootstrap one rump kernel.** The time varies from configuration to configuration because of the the initialization code that must be run during bootstrap.

3.13 Experiment: Bootstrap Time

Startup time is important when the rump kernel is frequently bootstrapped and "thrown away". This transitory execution happens for example with utilities and in test runs. It is also an enjoyment factor with interactive tasks, such as development work with a frequent iteration, as delays of over 100ms are perceivable to humans [38].

The bootstrap times for various rump kernel faction configurations are presented in Figure 3.30. In general, it can be said that a rump kernel bootstraps itself in a matter of milliseconds, i.e. a rump kernel outperforms a full operating system by a factor of 1000 with this metric. Furthermore, booting a rump kernel is faster than the time required for a hypervisor to launch a new instance. We conclude that a rump kernel boots fast enough.

www.dbooks.org

Network clusters

Bootstrapping a single node was measured to be an operation measured in milliseconds. High scalability and fast startup times make rump kernel a promising option for large-scale networking testing [24] by enabling physical hosts to have multiple independent networking stacks and routing tables.

We measure the total time it takes to bootstrap such a cluster in userspace, and to configure and send an ICMP ECHO packet through a networking cluster with up to 255 instances of a rump kernel. The purpose of the ICMP ECHO is to *verify* that all nodes are functional. The cluster is of linear topology, where node n can talk to the neighboring $n-1$ and $n+1$. This topology means that there are up to 254 hops in the network, from node 1 to 255.

We measured two different setups. In the first one we used standard binaries provided by a NetBSD installation to start and configure the rump kernels acting as the nodes. This remote client approach is most likely the one that will be used by most for casual testing, since it is simple and requires no coding or compiling. We timed the script shown in Figure 3.31. In the second setup we wrote a self-contained C program which bootstrapped a TCP/IP stack and configured its interfaces and routing tables. This local client approach is slightly more work to implement, but can be a valuable consideration if node startup and configuration is a bottleneck. Both approaches provide the same features during runtime. The results are presented in Figure 3.32.

The standard component approach takes under 8s to start and configure a networking cluster of 255 nodes. Although this approach is fast enough for most practical purposes, when testing clusters with 10-100x as many nodes, this startup time can already constitute a noticeable delay in case a full cluster is to be restarted. Assuming linear scaling continues, i.e. hardware limits such as available memory are not

```
#!/bin/sh

RUMP_COMP='-lrumpnet -lrumpnet_net -lrumpnet_netinet -lrumpnet_shmif'
[ $# -ne 1 ] && echo 'need count' && exit 1
[ ! $1 -ge 3 -o ! $1 -le 255 ] && echo 'count between 3 and 255' && exit 1
tot=$1

startserver()
{
        net=${1}
        export RUMP_SERVER=unix://rumpnet${net}
        next=$((${net} + 1))
        rump_server ${RUMP_COMP} ${RUMP_SERVER}

        rump.ifconfig shmif0 create
        rump.ifconfig shmif0 linkstr shm/shmif${net}
        rump.ifconfig shmif0 inet 1.2.${net}.1 netmask 0xffffff00

        if [ ${net} -ne ${tot} ]; then
                rump.ifconfig shmif1 create
                rump.ifconfig shmif1 linkstr shm/shmif${next}
                rump.ifconfig shmif1 inet 1.2.${next}.2 netmask 0xffffff00
        fi

        [ ${net} -ne 1 ] && \
            rump.route add -net 1.2.1.0 -netmask 0xffffff00 1.2.${net}.2
        [ ${next} -ne ${tot} -a ${net} -ne ${tot} ] && \
            rump.route add -net 1.2.${tot}.0 -netmask 0xffffff00 1.2.${next}.1
}

for x in `jot ${tot}`; do
        startserver ${x}
done

env RUMP_SERVER=unix://rumpnet${tot} rump.ping -c 1 1.2.1.1
```

Figure 3.31: **Script for starting, configuring and testing a network cluster.** This script can be used to test routing in up to the IP MAXTTL linearly chained TCP/IP stacks.

www.dbooks.org

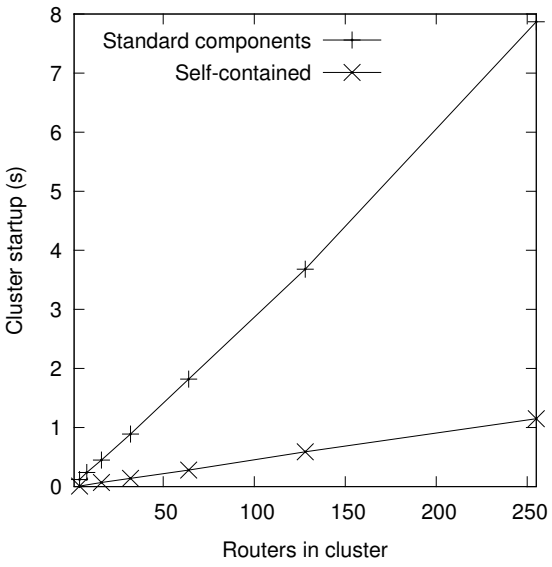


Figure 3.32: **Network cluster startup time**. The time is measured through starting the instances, configuring them, and up until the first packet sent from one end has made a full round-trip.

hit, the local client approach can bootstrap 10k nodes in 45 seconds, which is likely fast enough for all cluster reboot purposes.

3.14 Summary

We began this chapter by describing the cornerstone techniques for how to convert an existing monolithic kernel codebase into an anykernel. To retain the existing properties of the monolithic kernel, we did not introduce any new technologies, and adjusted the codebase using code moving and function pointers. These techniques were enough to convert the NetBSD kernel into an anykernel with an independent base and orthogonal factions.

We went over the various rump kernel implementation aspects such as implicit thread creation and the CPU scheduler. After that, we studied the effects that feature

relegation has on the implementation of the virtual memory subsystem and locking facilities. The rest of the chapter discussed various segments of the implementation, such as microkernel style file servers with rump kernel backends, USB hardware drivers and accessing rump kernels over the Internet.

We found out that many of the adjustments we did to NetBSD pertaining to the subject matter had a wider benefit. One example was the addition of the *ioconf* and *pseudo-root* keywords to a config file. This improvement simplified creating kernel modules out of device drivers and has been used by dozens of non rump kernel drivers since. Another modification we did was the ability to disable builtin kernel modules. This modification made it possible to disable drivers with newly discovered vulnerabilities without having to immediately reboot the system. These out-of-band benefits show that not only were our modifications useful in addressing our problem set, but they also benefit the original monolithic kernel.

www.dbooks.org

4 Rump Kernel Ecosystem

In the previous chapters we examined the core architectures of the anykernel and rump kernels. In this chapter, we will look at more practical aspects: how to build rump kernels on/for practically any architecture/OS, how to link rump kernels into software stacks and how to use the resulting software stacks. In other words, this chapter presents some use cases for rump kernels. As opposed to the previous chapter, we no longer limit the discussion to running rump kernels in NetBSD userspace.

We discuss software available from `http://repo.rumpkernel.org`.

4.1 buildrump.sh

To be able to run software, one must first compile software. Compiling software is done through a build framework, at least for any non-trivial project consisting of more than a handful of source modules. The rump kernel implementation grew around the native build framework of NetBSD. When building rump kernels for userspace as part of a NetBSD-targeted build, the right tools are automatically present and properly configured. Those tools are not automatically present in other situations, especially on non-NetBSD build hosts. The `buildrump.sh` script addresses the need of building rump kernel components on practically any POSIX-like host. The possible target platforms are legion.

At the core of `buildrump.sh` is NetBSD's intrinsic ability to cross-build itself on any platform. The build involves first building the build tools as host binaries, and then moving on to build the rump kernel binaries for the target system. The cross-build is accomplished by a script called `build.sh` [37]. After a fashion, the naming of `buildrump.sh` pays homage to the underlying script.

www.dbooks.org

The script is available from `http://repo.rumpkernel.org/buildrump.sh`, along with related subroutine scripts.

4.1.1 The Double-crossing Toolchain

Let us first consider what cross-building is. A cross-build can be from one OS or version to another, from one machine architecture to another, neither, or both. `buildrump.sh` always assumes the case of both, which means for example that it will not perform probes which require executing the target code.

The common case for the build host is an x86 Linux and the target is building rump kernel components for x86. In other words, in the common case we are building from one OS to another, but are building for the same machine architecture as the host. Therefore, the host's compiler knows how to generate target binaries. We will use this knowledge to optimize the user experience in the common case, while still supporting other cases as well.

The typical approach to cross-building is to first obtain a cross-toolchain and only after that proceed with the build. Contrary to standard practice, `buildrump.sh` does not build a toolchain. Instead, it creates wrappers around a user-supplied toolchain and builds the target binaries using those wrappers. The wrappers make the user-supplied toolchain look like a NetBSD toolchain, so that the NetBSD Makefiles work. For example, most compilers do not recognize the option `-cxx-isystem`. If the wrapper detects a compiler where that option is not supported, the option is translated to `-isystem` before the compiler is run.

The rationale behind using wrappers is convenience. First, downloading and building a cross-toolchain takes several times longer than building the actual rump kernel components. Second, since in the most common case (and a few others) there al-

ready is a usable toolchain sans wrappers, we would unnecessarily be burdening the user if we always required a NetBSD-targeting toolchain.

In cases where there is no possibility to use the host's toolchain, e.g. when on Mac OS X which uses a different object format (MACH-O vs. ELF), the user must obtain a suitable toolchain before running `buildrump.sh`. The same requirement for first having to obtain a suitable toolchain also applies when compiling to a different machine architecture, e.g. to ARM from an x86 host.

Some compilers may generate code for different machine architectures based on the supplied flags. For example, gcc targeting *x86_64* will generate code for 32bit x86 if `-m32` is passed as an argument. As part of its output, `buildrump.sh` will publish the wrappers which include the toolchain flags passed to `buildrump.sh`. So, for example, if an *x86_64* toolchain and `-m32` is passed to `buildrump.sh`, a `i386--netbsdelf` toolchain will be generated by `buildrump.sh`. In Section 4.2 we will look at how this set of wrappers can be used for further constructs on top of rump kernels.

4.1.2 POSIX Host Hypercalls

The first use case of `buildrump.sh` was to make building the rump kernel for Linux userspace practical and user-friendly. For the result to also be functional, a hypercall implementation was required. Due to that historical reason, the default mode of operation of `buildrump.sh` is to also build the POSIX hypercall implementation from `src-netbsd/lib/librumpuser` and also a handful of other libraries and utilities such as `src-netbsd/lib/librumphijack` and `src-netbsd/usr.bin/rump_server`.

Different POSIX'y platforms have subtle differences, for example in which POSIX version they conform to, and therefore what the exact set of supported inter-

www.dbooks.org

faces is. The first phase of building with a POSIX platform as the target runs a probe. The probe is a normal GNU autoconfigure script which is hosted in `src-netbsd/lib/librumpuser/configure`. The configure script checks for example if `clock_nanosleep()` is available, and therefore if it can be used to accurately implement `rumpuser_clock_sleep()` or if a best-effort use of `nanosleep()` is all that is possible.

When not building for a POSIX platform, the POSIX hypercall build must be explicitly disabled by the user. Disabling the POSIX hypercall build also disables the associated probe.

4.1.3 Full Userspace Build

Though the original purpose of `buildrump.sh` was to build kernel components, especially after work on the Rumprun unikernel (Section 4.2) began, it became clear that some easily invoked method for producing the corresponding NetBSD kernel and userspace headers and core userspace libraries, e.g. libc and libm, was required. This functionality was bolted on to `buildrump.sh`, and can optionally be invoked. It is worth taking the time to understand that producing full headers and libraries is orthogonal to building to run *on* a userspace platform.

4.1.4 src-netbsd

To build rump kernel components, `buildrump.sh` needs the source code for the relevant parts of the NetBSD tree. Theoretically, any vintage of the NetBSD source tree would work. However, in practice, a new enough vintage of the NetBSD tree is required. Historically, the user was required to obtain the NetBSD source tree before running `buildrump.sh`. Since the full NetBSD tree is large and since a

branch name	description
kernel-src	bare minimum sources necessary for building rump kernels. In addition to the kernel sources, the tools required for building rump kernels are also included in this branch.
user-src	various userspace libraries and utilities useful for common rump kernel applications, e.g. `libm` and `ifconfig`
posix-src	rumpuser implementation for POSIX platforms
buildrump-src	kernel + posix, i.e. what `buildrump.sh` builds by default
appstack-src	kernel + user, useful for e.g. unikernels (Section 4.2)
all-src	kernel + posix + user

Table 4.1: **src-netbsd branches**. The first set of branches are base branches which contain no overlap. The second set of branches are the convenience branches which contain a certain union of the base branches. You will most likely want to use a convenience branch for your project. The precise content lists for the base branches are available from `src/sys/rump/listsrcdirs`.

majority of the tree is not relevant for rump kernels, the relevant parts of the tree are now mirrored at `http://repo.rumpkernel.org/src-netbsd`. This source tree represents a known-good and tested vintage of the NetBSD source tree for use with rump kernels; the contents are a regularly updated snapshot of the NetBSD development head. The `checkout.sh` script in the buildrump.sh repository handles the details of the mirroring process.

The src-netbsd repository supplies several branches, with the idea being that the user can choose the minimal set of sources required for their particular application. The branches are listed in Table 4.1. These branches may be used as either a submodule, or fully duplicated into third party repositories.

www.dbooks.org

4.2 Rumprun Unikernel

The Rumprun unikernel is a unikernel [55] OS framework built on top of driver components provided by a rump kernels. Essentially, a unikernel is akin to an embedded system, where there is no separation between the application and system components of the software stack. Rump kernels are well-suited to building a unikernel framework, since the OS side of a unikernel is composed almost exclusively of drivers, as can be verified by examining the amount of non-driver code in Rumprun. While Rumprun is also applicable for bare metal embedded systems, the trend towards the cloud and microservices has made Rumprun particularly relevant due to the ability to run existing application-level programs.

The general idea of a unikernel is to bundle the operating system side components and application(s) into a single image. When that image is booted on a given platform, the instance is set up and the application is run according to the specified configuration. Notably, even though the application and system side components are bundled into a single image, the concept is orthogonal to whether or not both run in the same hardware protection domain.

The goal of Rumprun is to provide the simplest possible glue code for rump kernels to realize the target. We will discuss the inner workings of Rumprun. As usual, the exact details of how to build and configure the image are left to the user documentation.

The scope of the Rumprun unikernel is to enable creating bootable unikernel images from application-level source code. Those images may be manually launched to create instances. Details on how to deploy and manage a "fleet" of unikernel instances is beyond the scope of Rumprun. In other words, the *Orchestrating System* for the Rumprun unikernel is expected to be provided by 3rd parties.

As an option, Rumprun can provide a POSIX-like userspace environment which allows turning unmodified POSIX programs into unikernel images. An explicit goal of Rumprun was to enable existing POSIX'y code to run with zero modifications. That is not to say that everything will work without modification — consider that code occasionally must be ported to run between regular Unix platforms — rather that the ideal case will not require code modifications. Non-trivial applications and libraries, e.g. *libcurl*, *mpg123* and *sqlite*, do work without any changes. As we discuss Rumprun in this chapter, we will point out some limitations in Rumprun to further illustrate when code changes may be required.

A packaging system for tested and, where necessary, ported POSIX'y code to run on Rumprun is available at `http://repo.rumpkernel.org/rumprun-packages`. Further discussion on the packaging system is beyond the scope of this book.

The Rumprun unikernel works on top of x86 platforms on bare metal, Xen [3], KVM [29], and others. As of writing this, there is also nascent ARM support and Rumprun has successfully run networked services on an ARM evaluation board. In the following discussion, for machine dependent details, we will cover only x86, and more specifically x86_64.

Rumprun source code is available from `http://repo.rumpkernel.org/rumprun`. For an architecture diagram of the software stack, refer to Figure 4.1.

4.2.1 bmk – Bare Metal Kernel

Since a rump kernel is not a real kernel, we need a real kernel in our Rumprun software stack to provide functions which rump kernels do not give us. For review, these functions include for example bootstrap, thread creation, scheduling, interrupts and page level memory management. We call that kernel the *Bare Metal Kernel*, or *bmk*

www.dbooks.org

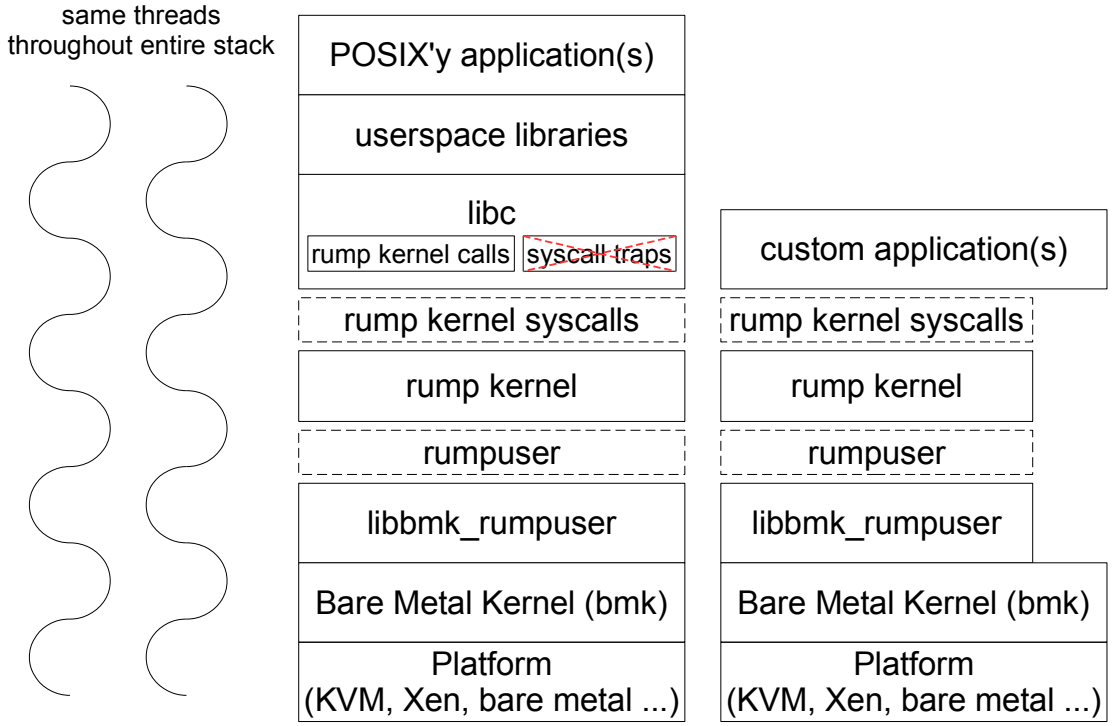


Figure 4.1: **Rumprun software stack.** Two modes are presented. The one on the left includes the POSIX'y userspace layers and can run unmodified programs. The one on the right is more lightweight, but mandates custom programs. POSIX'y programs are limited to POSIX'y interfaces, while custom applications may call the underlying OS layer directly. The specifics of the layers are discussed throughout this chapter.

for short. We will use the shorthand form from now on. bmk was originally written to demonstrate how to run rump kernels on top of bare metal. The more common recent use case is to run as a virtualized service on top of e.g. KVM, but the name stuck nonetheless.

The platform-specific implementations of bmk for Xen and non-Xen (e.g. KVM and bare metal) are different in places. We will limit our discussion to non-Xen platforms. The parts of the implementation common to all platforms can be found under the source tree in `lib/libbmk_core`. Platform-specific code is in `platform/xen` and `platform/hw` for Xen and non-Xen, respectively.

Bootstrap

The first stages of bootstrap are beyond Rumprun and relegated to a multiboot-compatible bootloader, provided by e.g. GRUB [22] or QEMU [48]. The assembly entry point for bmk is at `_start` in `platform/hw/arch/amd64/locore.S`. At the entry point, we save information provided by multiboot (e.g. memory size), set up bootstrap pagetables, switch the CPU to 64bit mode — a multiboot loader will leave the CPU in 32bit mode — set up the stack, and proceed to call the C entry point `x86_boot()`. In other words, we do the minimum of what more or less every operating system does at the early bootstrap stage.

The C startup code initializes the console and performs various architecture-specific initializations such as setting up the interrupt controller. Then, the scheduler is initialized, the page allocator is set up, and finally the main thread is created by calling `bmk_sched_startmain()`. The main thread eventually launches the application.

Threads and Scheduling

Recall, everything in a rump kernel runs in thread context (Section 2.3). On Rumprun, everything except the lowest-level interrupt acknowledgement runs in cooperatively scheduled thread context. The scheduler, including thread creation and TLS support, is implemented in `lib/libbmk_core/sched.c`. The Rumprun stack has no knowledge of processes, apart from what rump kernels provide.

The rationale for cooperative scheduling is that it produces not only a more readily repeatable result from one execution to another, but also ensures that a thread is never preempted unless it has finished its work and is ready to block. In fact, a cooperative thread scheduler matches the run-to-completion behavior of the rump kernel CPU scheduler. Therefore, the Rumprun unikernel will never encounter the

www.dbooks.org

situation where a host thread is preempted with the rump kernel context held, and therefore unnecessary host thread context switches are avoided.

In environments where the scheduler must protect against threads hogging all CPU time and preventing other threads from running, cooperative scheduling is not possible. Since in a unikernel all threads are essentially working towards the same goal, we can assume that there are no hostile threads. Of course, there is no reason why preemptive scheduling could not be implemented for bmk, just that we have chosen not to do so. Generally speaking, we are of the *opinion* that cooperative scheduling is more effective for the Rumprun unikernel, due to reasons mentioned in the previous paragraph.

The rump kernel interacts with the scheduler via the hypercalls, as usual. When a rump kernel hypercall encounters a situation where it decides to block, it unschedules the rump kernel context, sets up the conditions for the wakeup, and calls `bmk_sched_block()`. The scheduler then selects the next thread to run, or if none are available, blocks and waits for an interrupt to generate work. Blocking via the rump kernel happens completely transparently to POSIX-style applications using rump kernel system calls. Custom non-POSIX applications, in addition to blocking via a rump kernel system call, may also call the bmk scheduler directly. Figure 4.2 illustrates with a stack trace how a POSIX application interacts with the bmk scheduler. In that particular case, the wakeup will be generated by a timer interrupt; the timer interrupt will wake the interrupt thread, which in turn will wake up the application thread.

The cooperative scheduling model exhibits a feature over the typical preemptively scheduled pthreads. Any program with a thread which runs in a busy-loop without making blocking system calls will block the entire system. However, this type of behavior is not common and mostly found in programs for scientific computation. Nonetheless, one must be aware of the limitation when choosing the programs which

```
#0  bmk_sched_block ()                  [bmk]
#1  rumpuser_clock_sleep ()             [rumpuser]
#2  kpause ()                           [rump kernel]
#3  nanosleep1 ()                       [rump kernel]
#4  sys___nanosleep50 ()                [rump kernel]
#5  sy_call ()                          [rump kernel]
#6  sy_invoke ()                        [rump kernel]
#7  rump_syscall ()                     [rump kernel]
#8  rump___sysimpl_nanosleep50 ()       [rump kernel]
#9  __nanosleep50 ()                    [libpthread]
#10 _sleep ()                           [libc]
#11 main ()                             [application]
```

Figure 4.2: **Rumprun stack trace for `sleep()`**. The trace starts from the application and ends where bmk schedules another thread. The stack trace is annotated on the right with descriptions of which logical component each stack frame belongs to.

run on Rumprun. If it is absolutely necessary to run such programs, the best option is to insert yield calls into the busy-looping thread. To ensure correct execution of periodic tasks, application threads should yield or block dozens of times per second. Since yielding is cheap, it is better to err on the side of doing it more often than necessarily. Still, we want to stress that in our experience, regular I/O bound programs "just work" without modification.

4.2.2 Rumpuser

The rumpuser hypercall interface for the rump kernel is implemented on top of bmk. The implementation resides in `rumprun/lib/lib/libbmk_rumpuser`. The implementation can be used as a reference implementation for rumpuser especially for cases with underlying cooperative threading. We leave perusing the implementation to the interested reader.

www.dbooks.org

4.2.3 To Userspace (or Not To Userspace)

As indicated in Figure 4.1, two modes of operation are available: one which is capable of running POSIX'y userspace and one which is not capable of that. There are tradeoffs to including a full userspace stack in your unikernel instance. Throughout this text, by "userspace" we mean the normal userspace environment available on a regular POSIX'y operating system. On a unikernel, there is strictly speaking no hard division between the kernel and userspace, but we nonetheless use the term "userspace" to describe the POSIX'y application portion of the stack.

Like on a regular operating system, the userspace environment is a collection of library interfaces on top of which normal programs run. The system call interface is an analogous, lower level set of interfaces, but in most cases programs will run on top of the userspace environment, not directly on system calls. There are some exceptions, such as programs written in Go [54], where the language runtime is implemented directly on top of system calls.

The advantage of including userspace support is not only that POSIX programs work out-of-the-box, but also that userspace interfaces are mostly standard and stable. Therefore, no matter the future work we do on the Rumprun unikernel, we will always guarantee that userspace interfaces remain stable. We do not offer the same level of guarantee for bmk, even if we attempt to minimize churn.

The disadvantage of the full userspace stack is its extra footprint. Not only are userspace libraries mandated, but in practise the rump kernel file system components are needed because of various libc interfaces implicitly assuming the presence of certain files, e.g. `/etc/services` and `/etc/resolv.conf`. Therefore, if you need to minimize your footprint, and you do not have an existing, complex application written against POSIX interfaces, you most likely want to avoid the userspace layers.

```
#ifdef RUMP_KERNEL_IS_LIBC
__weak_alias(lseek,rump___sysimpl_lseek);
__weak_alias(_lseek,rump___sysimpl_lseek);
__strong_alias(_sys_lseek,rump___sysimpl_lseek);
#endif /* RUMP_KERNEL_IS_LIBC */
```

Figure 4.3: **Userspace aliases for rump kernel syscalls**. Conditionally, a rump kernel can provide system calls with names that a userspace environment expects. Both user-visible overridable (weak) and system-internal non-overridable (strong) aliases are provided. Not all aliases are necessary for all system calls, but since they do no harm, we provide them. For discussion on the system call entry point itself, see Section 3.6.1. As usual, there is a naming problem (with the macro name), but since the name is not user-visible, it has not been worth the fuss to adjust the name.

System Calls

We know from Section 3.6.1 that a rump kernel provides ABI-identical system calls apart from a `rump_sys_` prefix in the symbol name. We also implicitly understand that some component in the Rumprun software stack must provide the system calls under the same name as a regular libc. Furthermore, performing a system call must result in the handler in the rump kernel being invoked.

The system call entry points in libc invoke the kernel via a hardware trap. Those entry points may be useful in the future if we wish to run the application portion and system portion of the Rumprun stack in separate protection domains. However, in the simplest model we do not wish to assume anything about the underlying platform's capability to support privilege levels, and therefore the standard libc entry points are not applicable.

We augment the rump kernel system call handlers to conditionally alias the rump kernel entry point symbol to the libc symbols. These aliases are illustrated in Figure 4.3. For the Rumprun unikernel, we build the rump kernel with that conditional knob turned on. Furthermore, we build libc without the usual trap-generating entry

www.dbooks.org

points. When everything is linked together, an application unaware of rump kernels calling `foo()` results in the same as an application aware of rump kernels calling `rump_sys_foo()`. In effect, POSIX applications on Rumprun act as local rump kernel clients without being aware of it.

POSIX Threads (pthreads)

The POSIX threads or pthreads library offers a set of interfaces upon which multi-threaded applications can be realized. Those interfaces include for example ones for creating threads and performing synchronization between threads. Given the standard nature and widespread use of pthreads, we wish to support programs which use pthreads.

Again, there are multiple straightforward ways on how to realize pthread support. One is to write a pthread library from scratch. Another one is to port a pthread library from another operating system. However, both of those approaches incur implementation effort and maintenance, and are against our general principles of design.

We observe that the NetBSD pthread library is implemented 1:1 on top of NetBSD's kernel threads, i.e. the relation between an application pthread and a kernel thread is a bijection. To for example create a new thread, libpthread uses the `_lwp_create()` system call, and to put the current thread to sleep awaiting wakeup, `_lwp_park()` is called.

To support the NetBSD pthread library on top of the threads offered by bmk, we implemented the `_lwp` interfaces against bmk in `lib/librumprun_base/_lwp.c`. After that, we could use NetBSD's libpthread to provide pthreads on Rumprun.

As an implementation detail, as of writing this, `_lwp.c` is implemented as libc-level symbols in the userspace package (rumprun_base). Henceforth, software wanting to bypass libc and use the lwp interfaces to implement their own threading, e.g. the Go runtime, must include userspace support. This feature may be fixed at a future date by pushing the implementation of the lwp interfaces into the rump kernel.

Limitations

Recall, rump kernels do not support virtual memory or preempting threads. Therefore, rump kernels do not provide memory mapping (`mmap()`, `madvise()` and friends) or signals. These two facilities are used by some userspace applications.

For signals, we simply resort to stating "signals are evil". (There are advantages to this book no longer being an academic text.) Therefore, any application requiring signals for basic functionality will not work without porting. Coming up with mostly functional signal emulation, where handlers are called at select points without preempting threads, may be done at a future date. Regardless of whether that will be emulated or not, signals still are evil.

We emulate some aspects of memory mapping in the component library located at `lib/librumpkern_mman`. Notably, emulation is hooked in as rump kernel syscalls so that custom applications may use that emulation. Anonymous memory mapping is simply a matter of allocating the right amount of memory, though Rumprun will not respect the read/write/execute protection flags. Memory mapping files is more complicated, since the contents need to be paged in and out. Since there is no virtual memory, there are no page faults, and contents cannot be paged in on demand. As of writing this, read-only mapping are emulated by reading in the full contents at the time of the mapping. While the approach is not perfect in many ways, it allows a decent set of programs to work in some cases at the cost of a handful lines of code.

www.dbooks.org

4.2.4 Toolchain

The Rumprun unikernel is always cross-compiled, which means that the build process never runs on a Rumprun instance. Instead, the build process runs on a build host, e.g. a regular Linux system. To [cross-]build software, a toolchain is required.

For custom applications, we have no external standards to bow down towards, nor do we want to impose any limitations on how to build custom applications. Therefore, it is up to the builder of the custom application how to build and link the unikernel image. A straightforward way is to use the toolchain wrappers generated by `buildrump.sh` (Section 4.1.1). We will not further discuss toolchains for custom applications.

For POSIX'y userspace applications, we do have external standards to bow down to. Application are engineered to build using a build system (e.g. GNU autotools or CMake). Build systems assume that the toolchain looks a certain way, and that the build consists of certain steps. Those steps can be for example, probe, build and link. Recall, in the best case scenario unmodified applications work on a Rumprun unikernel. It would be convenient if those applications could also be built for the Rumprun unikernel without having to introduce changes to the build system. That was the goal of the application toolchain. The rest of the discussion in this section is on how that goal was accomplished.

The ABI

When building application source code, one must decide which ABI to build for. That ABI will determine where the program can run. Typically, the ABI is signified by the *tuple* ingrained into the toolchain. For example, a `x86_64-linux-gnu` toolchain will produce a binary which can run on an x86-64 machine with a Linux

kernel and a GNU userspace. In other words, there is a machine architecture component and a system side component to the ABI. The binary can be run only on a machine architecture and an operating system which supports the given ABI. The obvious example of an operating system supporting the Linux-GNU ABI is Linux, but it is also possible to other operating systems to emulate various ABIs to allow running programs complied for a non-native ABI.

For Rumprun, we use the same tuple as NetBSD, e.g. `x86_64--netbsd`, but insert "rumprun" into the otherwise empty vendor field. Therefore, for x86_64 the ABI tuple of Rumprun is `x86_64-rumprun-netbsd`. The installed toolchain for building Rumprun images follows the standard convention of naming binaries, e.g. `x86_64-rumprun-netbsd-ar` and `x86_64-rumprun-netbsd-gcc`. Internally, the toolchain is a set of wrappers around the toolchain provided by `buildrump.sh`, which in turn is a set of wrappers around the toolchain supplied by the user.

The Implementation of the ABI

In the normal case, the operating system implementing the system side of the ABI is not bundled with the binary. The operating system itself comes to be when a certain selection of drivers implementing that ABI is booted. Notably, there is no strict contract between the ABI and which drivers the operating system must provide for the program to run correctly. A Linux kernel without networking support can still run Linux binaries, but programs using networking will not run as expected on that particular Linux instance.

In other words, in the normal model, building & booting the operating system and building & running applications are separate steps. Normal build systems also assume them to be separate steps, and do not include a step to determine which operating system components should be linked into the binary. If we wish normal

www.dbooks.org

build systems to work without modification, we must address this disparity on our side.

Pseudo-linking and Baking

One solution for specifying the implementation of the ABI would be to hardcode the set of rump kernel components into the toolchain and to identify the set in the toolchain tuple. However, that solution would mean creating a separate set of toolchain wrappers for every desired component combination, and would be hard to manage. Another option would be to always include all drivers, but it would be wasteful, and also possibly would not work — consider situations where you have two mutually exclusive drivers for the same backend.

A more flexible solution comes from what we call *pseudo-linking*. When the application part of the binary is linked, the operating system components are not attached to the binary. Instead, the application link phase produces non-runnable intermediate format, which includes the objects and libraries specified on the link command line.

The pseudo-link phase checks that all application symbol dependencies are satisfied. Again, doing so honors existing conventions; this check is required for example by the probe phase of some build frameworks, which determine if a certain interface is supported by the given system by trying to link a test program using that interface, and iff the link succeeds, mark that interface as available. It needs to be noted that these types of checks apply only to userspace interfaces, and not to the kernel drivers backing those interfaces, so things may still fail at runtime. However, limiting the check to userspace interfaces is what current POSIX'y applications expect.

To produce the bootable unikernel image, the pseudo-linked intermediate repre-

```
$ cat > hello.c << EOF
> #include <stdio.h>
> int main() {printf("Hello, Rumprun!\n");}
> EOF
$ x86_64-rumprun-netbsd-gcc -c -o hello.o hello.c   # compile
$ x86_64-rumprun-netbsd-gcc -o hello hello.o        # pseudo-link
$ rumprun-bake hw_virtio hello.bin hello            # bake
```

Figure 4.4: **Building a runnable Rumprun unikernel image**. The different phases of the build are illustrated. First, the object files are compiled. Second, the object files are pseudo-linked. Any objects and libraries specified at this stage will be carried by the intermediate representation. Finally, the remainder of the Rumprun stack is baked into the pseudo-linked intermediate representation. In this example, we used the *hw_virtio* configuration, which contains the I/O drivers for cloud hypervisors. In case *hw_generic* is specified instead, drivers for bare metal would be included.

sentation is *baked*. The baking process attaches the operating system side driver components to the image, analogous to the normal case where the operating system implementing the ABI is booted. Baking is done using the `rumprun-bake` tool.

The entire set of steps require for transforming source code to a bootable unikernel image is illustrated in Figure 4.4.

Pipelines and Multibaking

Consider the Unix pipeline: a program generates output which is fed to the next stage in the pipeline as input. Therefore, a program does not need the knowledge of how to generate the input or how to store the output, as long as the program has the knowledge of how to process the data. Assume we have such a program which knows how to process data, but not whence the data is coming or where it is going to. We wish to support that concept in Rumprun, for example for cases where POSIX'y programs are used as highly isolated data processors.

www.dbooks.org

For example, imagine that the user wishes the main program to read input using HTTP, but the program lacks intrinsic HTTP support. This feat would be accomplished on a regular system with e.g. "`curl http://add.ress/input | prog`". There are a few possible routes to support the same in Rumprun. We will discuss some of those possibilities before discussing the option we chose.

- **adjusting the main program**: our guiding principle throughout this entire work is to avoid unnecessary forking and modification. In some cases teaching the main program may be necessary, for example when processing multiple small files is desired. However, for this discussion we will consider adding knowledge of HTTP to the main program as a last resort.
- **use of a file system driver**: a file system is akin to a system-side pipeline, though possessing the additional features of named streams and seeking, among others. It would be possible to hide the details of HTTP from an application inside a file system driver. However, due to the above-mentioned additional features which are usually not not required in a pipeline, file system drivers require implementing a dozen-or-so methods to have even bare-bones functionality. Therefore, the implementation effort of what could be accomplished with a handful of lines of application code is increased by at least an order of magnitude.

To solve the pipeline problem, we observe that a rump kernel already supports multiple processes, as required by remote client support (Section 2.5), and pipes between those processes (`rump_sys_pipe()`). Using a regular pipe between the rump kernel processes allows data to flow between the programs. The remaining puzzle pieces come from the separation of pseudo-linking and baking. We allow `rumprun-bake` to take multiple binaries which are baked into the final executable. For example, the following command will include both the processor and transporter in the final runnable binary:

```
$ rumprun-bake hw_virtio rumprun.bin processor transporter
```

This unikernel can then be run so as to imitate the following pipeline:

```
$ ./transporter r src | ./processor - | ./transporter w dst
```

Note, in the current implementation the transporter and processor share for example the same libc state and symbol namespace. Therefore, as of writing this, multibaking cannot be used to include multiple arbitrary programs in the same Rumprun image. However, the approach works in cases where the transporter program is judiciously crafted to suit the purpose. Support for arbitrary programs may or may not be added at a future date.

4.2.5 PCI: Big Yellow Bus

In Section 3.10 we looked at using USB devices in rump kernels. Ignoring the desirability of that approach, for Rumprun the same approach of relying on a thick underlying layer is not feasible — the only underlying software layer is bmk. For Rumprun on x86 hardware(/KVM), the desired peripheral I/O devices are on PCI busses. To use those devices, we provide PCI bus access via hypercalls, and use the PCI device drivers provided by the rump kernel for actual device access. The hypercalls follow the boundaries of the MI/MD (machine in/dependent) code boundary in NetBSD. We stress that the discussion in this section is about x86 bare metal.

Essentially there are three classes of hypercalls that are required to support PCI devices:

www.dbooks.org

- **PCI configuration space access**. By accessing configuration space registers, software can determine which devices are present, alter PCI-specific behavior of the device, and gain access to the device-specific registers. Configuration space access is done using x86 I/O operations (`inl()`, `outl()`). While those instructions could be used within the rump kernel without resorting to hypercalls, we use hypercalls for the benefit of being able to share the PCI code with other platforms which are not described here.

- **Interrupt handling**. Establishing an interrupt means that when an interrupt is triggered on the CPU, the corresponding interrupt handler gets called to inspect the device. While the process is simple in theory, a problem arises from mapping the device's idea of an interrupt line to the corresponding line detected by the CPU. Most systems attempt to establish this relationship by parsing the relevant ACPI tables. We take a simplistic approach which does not require thousands of lines of code: since there are typically only 1-2 PCI I/O devices on a given Rumprun instance, we assume that all devices share an interrupt and call all handlers when any PCI interrupt arrives. We may include proper support for interrupt routing at a future date, if a need arises.

- **DMA-safe memory handling**. DMA is not a PCI-specific construct, but since PCI devices are the only devices on Rumprun which do DMA, we discuss DMA here. Essentially, DMA is a matter of being able to allocate "DMA-safe" memory. In practice, safety means physically contiguous memory which is allocated according to certain boundary and alignment constraints. The boundary constraint means the range must not cross a certain multiple, and alignment means that the range must start from the multiple. Since bmk uses 1:1 mapped memory, any contiguous address range is also physically contiguous.

  In theory, NetBSD PCI drivers may request to allocate DMA memory in multiple physical segments and then map those segments to a virtually contiguous range in the kernel VA. A general solution for such a mapping needs

virtual memory support; however, we have not yet come across drivers which would use this functionality, and hence have not felt a need to solve the unencountered problem.

4.3 rumpctrl

A normal operating system is configured for operation and inspected at runtime by a selection of userspace utilities. Examples of these utilities on NetBSD include `ifconfig`, `mount`, `sysctl` and `raidctl`. Those utilities are useful for controlling and monitoring the runtime state of for example the Rumprun unikernel. From our discussion in Section 3.12, we know that we can run such binaries as remote clients. However, the problem arises from where to host such binaries.

On a NetBSD system, the right set of utilities are available as regular host binaries, and we can run those binaries as hijacked remote clients (Section 3.12.6). The system calls and data structures used by the utilities are non-portable, and mechanisms such as `ioctl()`, `sysctl()` and *the routing socket* are used. Therefore, it is highly non-trivial to port and compile those utilities on other operating systems. Furthermore, the system call interfaces used by the utilities evolve as the capabilities of the kernel drivers evolve, and therefore it is not enough to port the utilities once. In effect, the utilities cannot be used on non-NetBSD systems and porting the utilities to those systems involves the porting work and continuous maintenance work.

One possibility would be to require running the control utilities on a NetBSD system, perhaps in a virtual machine. However, that would be against our principle of convenience for the user.

The solution comes from running a unikernel-like stack in userspace coupled with remote system calls (`librumpclient`). Notably, no hijacking of system calls is

www.dbooks.org

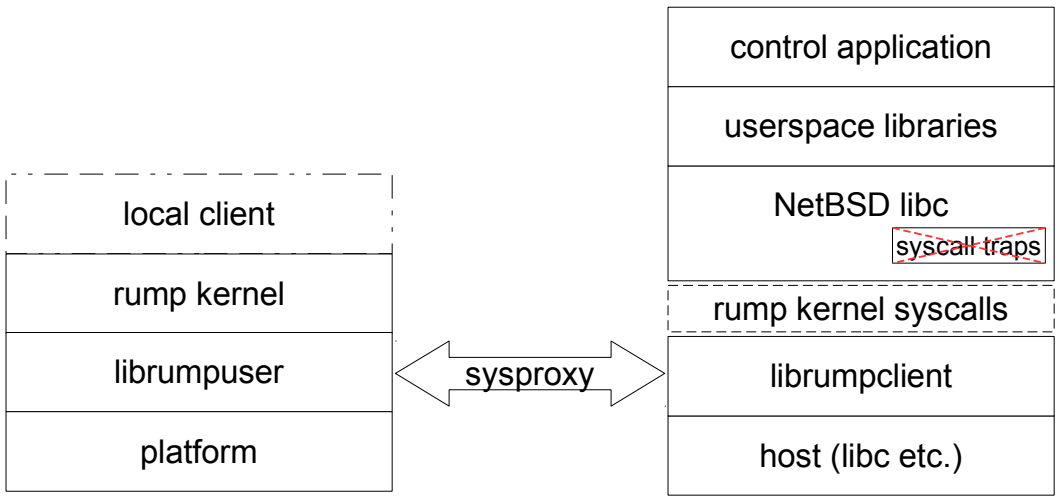


Figure 4.5: **Architecture of rumpctrl**. The stack on the left is the rump kernel stack being controlled. That stack may or may not include a local client. For example, a Rumprun unikernel instance is likely to include a local client, while `rump_server` (Section 3.12.3) will not. The stack on the right runs in POSIX userspace, and communicates with the rump kernel over a sysproxy transport.

required since system calls are directed against the rump kernel by default, just like in the case of Rumprun. To avoid collisions between the host's libc and the NetBSD libc, careful symbol renaming is performed during the build stage. The architecture of the solution is depicted in Figure 4.5.

The usage of rumpctrl follows the same principles as remote clients (Section 3.12.2); the environment variable `RUMP_SERVER` contains the URL which points the client to the server. Additionally, rumpctrl provides a source'able script which sets the rumpctrl commands at the front of `PATH`. In the following demonstration we have a rump kernel server (in this case a Rumprun unikernel) listening to sysproxy commands on a management interface at 10.0.0.2:12345. Additionally, we demonstrate the `rumpctrl_listcmds` command, which prints the commands provided by rumpctrl.

```
$ . rumpctrl.sh
rumpctrl (NULL)$ sysctl hw.model
error: RUMP_SERVER not set
rumpclient init failed
rumpctrl (NULL)$ export RUMP_SERVER=tcp://10.0.0.2:12345
rumpctrl (tcp://10.0.0.2:12345)$ sysctl hw.model
hw.model = rumpcore (virtual)
rumpctrl (tcp://10.0.0.2:12345)$ rumpctrl_listcmds
arp             ed              mkdir           newfs_ext2fs    rndctl
cat             fsck            mknod           newfs_msdos     route
cgdconfig       fsck_ext2fs     modstat         npfctl          rtadvd
chmod           fsck_ffs        mount           pax             sysctl
chown           fsck_msdos      mount_ext2fs    pcictl          umount
cp              halt            mount_ffs       ping            vnconfig
dd              ifconfig        mount_msdos     ping6           wlanctl
df              ktrace          mount_tmpfs     raidctl         wpa_passphrase
disklabel       ln              mv              reboot          wpa_supplicant
dump            ls              ndp             rm
dumpfs          mixerctl        newfs           rmdir
```

4.4 fs-utils

Fs-utils [60] (`http://repo.rumpkernel.org/fs-utils`) is a suite of userspace file system utilities (hence the name) which intrinsically contain file system drivers; the utilities do not use file system drivers from the host kernel. The motivations for building such a suite are the usual: running the file system driver in userspace does only not depend on having support in the host kernel, but also does not risk a host kernel compromise in case of a corrupt or maliciously corrupted file system image. In the permissions department, read and optionally write permissions to the image are enough, no elevated permissions are needed by the user running the utilities.

The implementation of fs-utils consists of standard standard NetBSD file utilities (`ls`, `cat`, `mv`, etc.) which use rump kernel file system drivers as local clients. Doing so preserves the normal usage of the utilities, e.g. `ls` accepts the familiar `-ABcFhL` parameter string.

www.dbooks.org

The only exception to command line arguments is that the first parameter is interpreted as the location specifier the file system is mounted from. The tools make an attempt to auto-detect the type of file system, so passing the file system type is optional. For example, `fsu_ls /dev/rwd0a -l` might list the contents of a FFS on the hard drive, while `fsu_ls 10.181.181.181:/m/dm -l` would do the same for an NFS export [10].

Alternative ways of implementing such a file system utility suite are to write or port the file system drivers, or use full operating systems in virtual machines. Those approaches are demanding in terms of programming work or runtime resources, respectively.

4.5 Summary

In this chapter we examined the ecosystem of tools and products built on top of rump kernels. At the center of the ecosystem is the `buildrump.sh` script, which allows building rump kernels on any POSIX-like system for a variety of targets. We discussed examples of what to build on top. One example was the Rumprun unikernel, which allows running POSIX'y applications on bare metal and the cloud. Another example was the fs-utils tool suite, which consists of tools capable of accessing and modifying file system images.

[10] In case of NFS, the *sockin* networking facility (Section 3.9.1) is used, so no TCP/IP stack configuration is required.

5 Short History

This chapter gives a short overview of the history of rump kernels and the events that lead to the current situation. The intention is to provide reasonable amounts of insight into why things evolved like they did. This chapter is written in chronological order on a high level, but we attempt to discuss one subject matter in one go, so there is no timeline in the strict sense.

We will use the late 2006 to early 2007 period as the starting point for the story. Back then, work was done on the *puffs* userspace file systems framework in NetBSD. This work led to key observation for rump kernels: it is very easy to implement a superficially working file server in userspace because a crash does not mean the end of everything. However, if the file server is supposed to be both performant and robust, things get very complicated even in userspace. In effect, the difference that userspace offers is the immeasurably nicer development environment.

The next logical question to raise was if there is any fundamental reason that kernel file systems cannot be developed in userspace without having to perform back-and-forth porting of the drivers between userspace interfaces and kernel interfaces. Given that you are reading this document, the answer is: no.

5.1 First Steps

In summer 2007, running NetBSD's kernel FFS driver as a puffs userspace server was made possible. The guideline for the work was that the FFS driver must remain unmodified, and various shims be built for linking and running the driver. Making the unmodified FFS driver work in userspace took approximately two weeks. A good part of that time was spent fixing bugs in the shim that translated file offsets

www.dbooks.org

to backing device block numbers (*getpages*, *putpages* & *bmap*). Translating offsets sounds simple enough, but when you have to take into account the sizes of device blocks, file system blocks and memory pages, along with potentially extending the file, things start getting complicated.

Originally, the working name for rump kernels was *sakern*, which was officially supposed to mean "standalone kernel". Unofficially, it was supposed to be a reference to the Swedish language and mean "things", denoting the slightly messy nature of the shims. The name was changed to *RUMP* before import to the NetBSD source tree. Later, that "backronym" would be dropped in favor of "rump kernels". For consistency, we will simply use "rump kernels" throughout this section, instead of altering the nomenclature based on which timeframe we are talking about.

In August 2007, rump kernels were introduced into the NetBSD source tree. At that time, there was support for a number of file systems which could run as puffs servers. Support for running the file drivers as standalone programs was yet not available, since there was no support for the top layers of the kernel. For example, the *namei* part of the *namei+lookup* ping pong game was handled by puffs in the host kernel, and rump kernels only supported *lookup* performed by individual file system drivers.

Also around the same time, the first steps were taken to run NetBSD kernel code on non-NetBSD hosts. Some assumptions about the host being NetBSD were addressed and fixed, and a mental note was made about what was required for supporting non-NetBSD platforms. At that time, building for non-NetBSD required huge amount of manual work, and it would not be until late 2012 when *buildrump.sh* was introduced that building any given vintage of rump kernels for non-NetBSD platforms would be made easy.

5.2 Towards Robust and Maintainable

Hacking up something that runs is not difficult. The real challenge is in making things maintainable in a system under constant development. While the shims made it possible to run kernel drivers from one vintage of NetBSD, the shims would eventually go out-of-date and cause compile-time and runtime failures. Since NetBSD kernel code will not go out-of-date with itself, shims were slowly replaced by using NetBSD kernel code directly. Figuring out which interfaces need to implemented as shims, and which interfaces can be used directly from NetBSD sources ended up being a multi-year effort. The general principles on what to reuse and what to rewrite have been more or less clear since 2011 and the core technology has not changed since then.

Another area that needed improvements was multithreading support. Initially, rump kernels did not support locking, synchronization, or anything that interacted with the scheduler. This was fine for file system drivers, because most of them use only a single thread, and simply ignoring most of the aspects of locking was enough. The problem with that approach was of course that it was not good enough to support any given kernel driver, and neither could it be used to exercise the multithreading robustness of file system drivers.

Most of the support for multithreading was added later in 2007. Support for kernel threads and locking and was relatively straightforward to add. The hard part of multithreading support was figuring out how the inverse scheduling model should be handled — normally operating systems pick a thread to schedule on a core, but rump kernels schedule a virtual core for a thread. The original design of the scheduler was done on an airplane napkin in 2009, but coming up with a deadlock free implementation was anything but trivial. It took over a year to successfully implement the design.

www.dbooks.org

5.3 Syscall Support

As mentioned, in the beginning rump kernels supported only file servers, and those accessed the kernel drivers directly at the in-kernel virtual file system layer. Hence, there was no need to support the POSIX-style system call interfaces.

However, as application possibilities such as *fs-utils* were being imagined, the necessity for a syscall-like interface grew. One early attempt at a syscall-like facility was *ukfs* (or "userspace kernel file system"). It implemented enough of the top half of the kernel for syscall-like pathname-based access to file systems to be possible.

While ukfs worked for file systems, it had issues:

1. The implementation of the top half, especially *namei*, is complex, so getting it right was non-trivial.
2. The interface was not quite syscall-like, which necessitated writing applications specifically for ukfs instead of being able to use existing applications against rump kernels in a library-like fashion.
3. ukfs was specific to file systems and did not address other subsystems such as the networking stack.

To remedy this, the top levels of the kernel, including a subset of the system call handlers and namei, were added to rump kernels in the beginning of 2008. One mistake was originally made with the introduction of the rump kernel syscall interface. To avoid the implicitly thread-local quality of the `errno` variable, each system call interface in the rump kernel syscall API took an extra "`int *error`" parameter for returning the error. This approach made it difficult to use existing applications against the rump kernel syscall API, since the application side required modification. It took almost a year for work to progress far enough for the extra parameter

to become a problem. The extra error parameter was removed in January 2009 now making the rump kernel syscall API and ABI equal to the one provided by the NetBSD libc.

As for ukfs, the homegrown pieces were removed, and the interface was implemented on top of rump kernel syscalls. For some years, ukfs was intended to become a portable interface for accessing file systems, i.e. one which does not suffer from type incongruence between the rump kernel and the host (think of the canonical "`struct stat`" example). However, ukfs is now considered all but an obsolete early experiment on the road of figuring out how to best do things.

5.4 Beyond File Systems

From the start, the idea was to support multiple kernel subsystems with rump kernels. This is evident also from the directory structure of the initial 2007 import: file systems were in `sys/rump/fs`.

It took over a year from the initial import in 2007 for support for another subsystem to appear. The natural choice was networking, since it was required by NFS, and support was added in late 2008. As part of the initial networking support, a virtual network interface to access the network via the host's *Ethernet tap* device was added. The idea was to provide network access, not to necessarily be fast. Support for the fast userspace packet I/O frameworks DPDK and netmap emerged in 2013 — not that those frameworks even existed back in 2008.

Beyond file systems and networking, the remaining major source of drivers is for devices. These were the last major set of drivers for which rump kernel support was added. The first steps were taken in mid-2009 with the support of the kernel autoconfiguration framework in rump kernels. Later in 2009, the first experiments

www.dbooks.org

with driving actual devices was taken in the form USB devices. Even so, the USB implementation did not touch raw devices, and depended on the USB host controller driver being supported by the host. Support for real device drivers appeared first in 2013 with PCI device support under Xen.

5.5 Symbol Isolation

If code is compiled and linked into a non-freestanding environment, there is a risk of symbol collisions. For example, consider hosting rump kernels in userspace: both the rump kernel and host libc provide a function called `malloc()`. However, the kernel malloc of NetBSD has a different signature, and takes e.g. a flag parameter which specifies whether or not the returned memory should be zeroed. In early versions of rump kernels, kernel and userspace symbols were haphazardly linked together. For example, if the kernel malloc is serviced by the libc implementation, the flags will be ignored and potentially non-zeroed memory will be returned. Amusingly enough, it took a few months before a bug was triggered by malloc not zeroing memory even if it was expected to.

Early attempts at avoiding symbol collisions were done in an ad-hoc fashion. For example, malloc was dealt with by using the `--wrap` flag in the linker, and for example the kernel `valloc()` routine was renamed the `vnalloc()` to avoid a collision with userspace. The exact set of conflicting symbols, however, depend on the platform, so the early attempts were never considered anything more than bandaid.

The real solution came in early 2009 with mass symbol renaming: at build-time, all symbols in a rump kernel are mass renamed to start with the prefix "rump". While doing so solved the collision problem, the renaming also had a much more profound implication. After symbol renaming was instated, the rump kernel was now a closed namespace; a rump kernel could only access routines within itself or in the hypercall

layer. We now knew for certain that a rump kernel would not block without making a hypercall. This knowledge led to understanding the required scheduling model. Furthermore, hypercalls now defined the portability interface for rump kernels.

5.6 Local Clients

One of the things to figure out about the scheduling of rump kernels was where and how to decide which process and thread context the currently executing host thread possesses in the rump kernel. Originally, and heavily influenced by the original file server mode of operation, it was completely up to the client to specify the PID and thread ID. Having the client select the PID allowed for the file server to operate with the same process context as the original operation in the host kernel. Having the same PID meant that errors produced by the fs driver in the rump kernel, e.g. "file system full", reported the correct host PID that had caused the malfunction.

The downside of the "client decides" approach was that the client had to decide even if it did not want to, i.e. there was no simple way for the client to say "I want a new process for a new set of file descriptors, don't care about the PID". The end result of all of this was that emulating things like fork and exit for remote clients would have been close to impossible.

Figuring out a sensible interface for deciding the current context took many years of development and 3 or 4 rewrites of the interface. In 2010, the *lwproc* set of interfaces was introduced, along with the concepts of implicit and bound threads, marking a big milestone in defining the unique characteristics of rump kernels.

www.dbooks.org

5.7 Remote Clients

The first experiments with remote clients were done in early 2009. However, it took over 1.5 years for a revised implementation to appear. Unlike in the case of the scheduler, the delay was not due to the code being complicated to figure out. On the contrary, it took so long because writing RPC code was boring.

While remote clients showed that it is possible to use Unix as a distributed OS without requiring a complete Plan 9-like implementation from scratch, that was not the main goal of remote clients. Support for remote clients made it possible to configure a rump kernel in a more natural way, where a configuration program is run with some parameters, the program performs some syscalls on the kernel, after which the program exits. Before remote clients, any configuration code was run in a local client, usually written manually. This was not a huge problem for file systems, since the configuration is usually just a matter of calling `mount()`, but for configuring the network stack, not being able to use `ifconfig`, `route` and friends was becoming an issue.

5.8 Shifting Focus Towards Driver Reuse

In early 2011, rump kernels were declared complete; the basic infrastructure required by portable kernel drivers had been figured after about four years more of more or less full-time work. The declaration also marked the end of most of the work taking place in the NetBSD source tree, since the main architecture of rump kernels had been defined. Given that the goal is to stick to unmodified NetBSD source code, a decent amount of work still goes on directly in the NetBSD tree, but other development loci are emerging.

The first major landmark was the introduction of the *buildrump.sh* script in late 2012. The importance of buildrump.sh was in that for the first time it was simple to build a given NetBSD source vintage for a non-NetBSD platform. Buildrump.sh was also the first piece of code to be committed onto GitHub to what would eventually become the rumpkernel organization.

The shift toward driver reuse also brought a completely new focus for development. Earlier, when debugging was the main motivation, the main focus was making rump kernels convenient. A good example of this attitude are the implicit threads provided by the rump kernel scheduler: if you do not know about bound threads or do not care to manage them, the default implicit threads will always work correctly, though they do not perform well. With driver reuse, and especially with networking, it started being important to be as performant as possible. Improving performance is still on-going work, but steps such as improving the performance of `curlwp` have been done in 2014. In fact, the old and new goals conflict slightly, because with implicit threads things will work just fine, but unless you know about bound threads, you will be losing a good deal of potential performance.

5.9 We're not in Kansas Anymore

Up until late 2012, rump kernels were only able to run in userspace processes. Support was mostly limited to NetBSD and Linux at the time, but it's a small hop from one userspace to another (though the amount of details involved in those small hops is rather dumbfounding). The challenge was to evaluate if rump kernels could run on top of literally anything.

The first non-userspace platform ended up being a web browser, with kernel drivers being compiled to JavaScript instead of the usual assembly. That experiment was both a success and a failure. It was a success because rump kernels ran smoothly in a

www.dbooks.org

web browser when compiled with a C→JavaScript compiler. It was a failure because the compiler provided emulation for the POSIX interfaces, and the existing rump kernel hypercalls could be used. In essence, a rump kernel running as JavaScript in a web browser still worked more or less like when running in userspace.

The second non-userspace experiment was undertaken in early 2013. It was a bit more successful in choosing a decidedly non-POSIX platform: the Linux kernel. There were no big surprises in getting rump kernels running on the Linux kernel. One interesting detail was the i386 Linux kernel being compiled with `-mregparm=3`. That compiler parameter controls how many function arguments are passed in registers, with the rest being passed on the stack. If the rump kernel components are not compiled with the same parameter, the ABIs do not match, and chaos will result. The same concern applies to routines written in assembly. The discrepancy is not insurmountable, merely something that needs to be noted and addressed where necessary. Another useful thing that came out of the Linux kernel experiment was minor improvements to the rumpuser hypercall interface. The hypercall interface has remained the same since then.

Even though the Linux kernel is not a POSIX environment, it still resembles userspace a fair deal: there are scheduled threads, synchronization primitives, etc. To further test the portability of rump kernels, a hypercall implementation for the Xen hypervisor was written in summer 2013. Since the Xen hypervisor is a bare-bones platform without the above mentioned userspace-like synchronization and multiprocessing features, it would test rump kernel portability even more than the Linux kernel did. Granted, the hypercalls were written on top of the Xen MiniOS instead of directly on top of the hypervisor, but MiniOS is there only to provide bootstrapping support, access to paravirtualized devices and MD code for stack switching. All in all, that platform was more or less like running rump kernels directly on top of bare metal.

The Xen platform was also the first to support running a full application stack on top of a rump kernel, including libc and other userspace libraries. Earlier, clients used rump kernels only as libraries, but especially standard applications depended also on the functionality provided by the host system. With additional work, it became possible to run off-the-shelf programs fully on top of rump kernels. The Xen platform was also the first time that a rump kernel was fully in control of the symbol namespace, instead of integrating into an existing one. Notably, not having to integrate into an existing namespace is technically simpler than having to, but it was useful to verify that rump kernels could cope nonetheless.

The first third party rump kernel hypercall implementation was done by Genode Labs for their Genode OS Framework. Support was first released as part of Genode 14.02, which in a highly mnemonic fashion was released in February 2014. Other third parties building OS and OS-like products have since started looking at rump kernels for driver support.

The final frontier, even if traditional operating systems it was the first frontier, was booting and functioning on bare metal. Support for x86 bare metal was written in August 2014. The implication of bare metal was also being able to support a number of virtualization platforms, e.g. KVM, which unlike paravirtualized Xen are bare metal except with alternative I/O devices. That said, the bare metal platform first supported only real I/O devices (e.g. the E1000 NIC instead of a virtio NIC), and virtual ones were added only later when the applicability for the cloud started becoming apparent.

The bare metal and Xen platforms were hosted in separate repositories for a long time, but eventually it became apparent that they consisted mostly of the same bits, and cross-porting changes from one to the other was becoming taxing. The repositories were combined in early 2015 and the resulting repository was called “Rumprun software stack”. The arduous process of merging the diverged code-

www.dbooks.org

bases was completed in April 2015. The same month the name was changed from "Rumprun software stack" to "Rumprun unikernel". For one, the change brought synergy with the growing unikernel trend, but the more important implication of using an established term was making it easier to explain Rumprun to newcomers.

5.10 Summary

The initial implementation of rump kernels built around file system drivers was minimalistic. Throughout the years the interfaces and implementation were modified to achieve both better client-side usability and maintainability. Introducing features and dependencies was avoided where feasible by critically examining each need. Eventually, stability was achieved.

The second phase was to show the value of rump kernels for providing production quality drivers. This phase was conducted by experimenting with using rump kernels as building blocks on various platforms.

6 Conclusions

A codebase's real value lies not in the fact that it exists, but in that it has been proven and hardened "out there". The purpose of this work is harnessing the power and stability of existing in-kernel drivers.

We defined an *anykernel* to be an organization of kernel code which allows the kernel's *unmodified* drivers to be run in various configurations such as libraries, servers, small standalone operating systems, and also in the original monolithic kernel. We showed by means of a production quality implementation that the NetBSD monolithic kernel could be turned into an anykernel with relatively simple modifications. The key point is retaining the battle-hardened nature of the drivers.

An anykernel can be instantiated into units which include the minimum support functionality for running kernel driver components. These units are called *rump kernels* since they provide only a part of the original kernel's features. Features not provided by rump kernels include for example a thread scheduler, virtual memory and the capability to execute binaries. These omissions make rump kernels straightforward to integrate into any platform that has approximately one megabyte or more of RAM and ROM. Alternatively, entirely new software stacks built around rump kernels are possible to execute with relative ease, as we explored with the Rumprun unikernel.

As the parting thoughts, we remind ourselves of why operating systems have the role they currently have, and what we should do to move forward.

The birth of timesharing operating systems took place over 50 years ago, an era from which we draw even the current concept of the operating system. Back then, hardware was simple, scarce and sacred, and those attributes drove the development

www.dbooks.org

of the concepts of the system and the users. In the modern world, computing is done in a multitude of ways, and the case for the all-encompassing operating system has watered down.

The most revered feature of the modern operating system is support for running existing applications. We can harness that power through rump kernels. Therefore, there is no reason to cram a traditional operating system into every problem space. Instead, we should choose the most suitable software stack based on the problem at hand.

References

[1] Slirp, the PPP/SLIP-on-terminal emulator. URL `http://slirp.sourceforge.net/`.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1992. Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. ACM Transactions on Computer Systems 10, no. 1, pages 53–79.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In: Proceedings of the ACM Symposium on Operating Systems Principles, pages 164–177.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A new OS architecture for scalable multicore systems. In: Proceedings of the ACM Symposium on Operating Systems Principles, pages 29–44.

[5] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pages 41–46.

[6] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1990. Lightweight Remote Procedure Call. ACM Transactions on Computer Systems 8, pages 37–55.

[7] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In: Proceedings of the USENIX Summer Technical Conference, pages 87–98.

www.dbooks.org

[8] Martin Campbell-Kelly. 1998. Programming the EDSAC: Early Programming Activity at the University of Cambridge. IEEE Annals of the History of Computing 20, no. 4, pages 46–67.

[9] Adam M. Costello and George Varghese. 1995. Redesigning the BSD Callout and Timer Facilities. Technical Report WUCS-95-23, Washington University.

[10] Charles D. Cranor. 1998. Design and Implementation of the UVM Virtual Memory System. Ph.D. thesis, Washington University.

[11] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. 1999. An Architecture for a Secure Service Discovery Service. In: Proceedings of the 5th MobiCom, pages 24–35.

[12] Luke Deller and Gernot Heiser. 1999. Linking Programs in a Single Address Space. In: Proceedings of the USENIX Annual Technical Conference, pages 283–294.

[13] Mathieu Desnoyers. 2009. Low-Impact Operating System Tracing. Ph.D. thesis, Ecole Polytechnique de Montréal.

[14] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-Level Implementations of Read-Copy Update. IEEE Transactions on Parallel and Distributed Systems 23, no. 2, pages 375–382.

[15] Edsger W. Dijkstra. 2001. My recollections of operating system design. URL http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF.

[16] Jeff Dike. 2001. A user-mode port of the Linux kernel. In: Proceedings of the Atlanta Linux Showcase. URL http://www.linuxshowcase.org/2001/full_papers/dike/dike.pdf.

[17] Aggelos Economopoulos. 2007. A Peek at the DragonFly Virtual Kernel. LWN.net. URL http://lwn.net/Articles/228404/.

[18] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. 1997. The Flux OSKit: A Substrate for OS and Language Research. In: Proceedings of the ACM Symposium on Operating Systems Principles, pages 38–51.

[19] Bryan Ford and Russ Cox. 2008. Vx32: Lightweight User-level Sandboxing on the x86. In: Proceedings of the USENIX Annual Technical Conference, pages 293–306.

[20] Ludovico Gardenghi, Michael Goldweber, and Renzo Davoli. 2008. View-OS: A New Unifying Approach Against the Global View Assumption. In: Proceedings of the 8th International Conference on Computational Science, Part I, pages 287–296.

[21] Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mary S. Weeks. 1987. Shared Libraries in SunOS. In: Proceedings of the USENIX Summer Technical Conference, pages 375–390.

[22] GNU GRUB. URL `http://www.gnu.org/software/grub/`.

[23] James P. Hennessy, Damian L. Osisek, and Joseph W. Seigh II. 1989. Passive Serialization in a Multitasking Environment. US Patent 4,809,168.

[24] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. 2008. Large-scale Virtualization in the Emulab Network Testbed. In: Proceedings of the USENIX Annual Technical Conference, pages 113–128.

[25] Jon Howell, Bryan Parno, and John R. Douceur. 2013. Embassies: Radically Refactoring the Web. In: Proceedings of the USENIX Conference on Networked Systems Design and Implementation, pages 529–546.

[26] Xuxian Jiang and Dongyan Xu. 2003. SODA: A Service-On-Demand Architecture for Application Service Hosting Utility Platforms. In: Proceedings

www.dbooks.org

of the 12th IEEE International Symposium on High Performance Distributed Computing, pages 174–183.

[27] Poul-Henning Kamp and Robert N. M. Watson. 2000. Jails: Confining the omnipotent root. In: Proceedings of SANE Conference. URL http://www.sane.nl/events/sane2000/papers/kamp.pdf.

[28] Antti Kantee. 2007. puffs - Pass-to-Userspace Framework File System. In: Proceedings of AsiaBSDCon, pages 29–42.

[29] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In: Proceedings of the 2007 Ottawa Linux Symposium, pages 225–230.

[30] Steve R. Kleiman. 1986. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In: Proceedings of the USENIX Annual Technical Conference, pages 238–247.

[31] Jochen Liedtke. 1993. Improving IPC by Kernel Design. In: Proceedings of the ACM Symposium on Operating Systems Principles, pages 175–188.

[32] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2010. System V Application Binary Interface, AMD64 Architecture Processor Supplement, Draft Version 0.99.5. URL http://www.x86-64.org/documentation/abi-0.99.5.pdf.

[33] Jim Mauro and Richard McDougall. 2001. Solaris Internals: Core Kernel Architecture. Sun Microsystems, Inc. ISBN 0-13-022496-0.

[34] Steven McCanne and Van Jacobson. 1993. The BSD packet filter: a new architecture for user-level packet capture. In: Proceedings of the USENIX Winter Technical Conference, pages 259–269.

[35] Paul E. McKenney. 2004. Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels. Ph.D. thesis,

OGI School of Science and Engineering at Oregon Health and Sciences University.

[36] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. 1996. The Design and Implementation of the 4.4BSD Operating System. Addison Wesley. ISBN 0-201-54979-4.

[37] Luke Mewburn and Matthew Green. 2003. build.sh: Cross-building NetBSD. In: Proceedings of the USENIX BSD Conference, pages 47–56.

[38] Robert B. Miller. 1968. Response time in man-computer conversational transactions. In: Proceedings of the Fall Joint Computer Conference, AFIPS (Fall, part I), pages 267–277.

[39] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. 1988. Kerberos Authentication and Authorization System. In: Project Athena Technical Plan.

[40] Ronald G. Minnich and David J. Farber. February 1993. The Mether System: Distributed Shared Memory for SunOS 4.0. Technical Report MS-CIS-93-24, University of Pennsylvania Department of Computer and Information Science.

[41] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. 1990. Amoeba: A Distributed Operating System for the 1990s. Computer 23, no. 5, pages 44–53.

[42] Mutt E-Mail Client. URL `http://www.mutt.org/`.

[43] NetBSD Kernel Interfaces Manual. November 2007. pud – Pass-to-Userspace Device.

[44] OpenSSH. URL `http://www.openssh.com/`.

[45] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, pages 1–16.

www.dbooks.org

[46] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. 1990. Plan 9 from Bell Labs. In: Proceedings of the Summer UKUUG Conference, pages 1–9.

[47] The Transport Layer Security (TLS) Protocol. 2008. RFC 5246.

[48] QEMU – open source processor emulator. URL http://qemu.org/.

[49] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. 1987. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. SIGARCH Computer Architecture News 15, pages 31–39.

[50] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. pages 101–112.

[51] Chuck Silvers. 2000. UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pages 285–290.

[52] A NONSTANDARD FOR TRANSMISSION OF IP DATAGRAMS OVER SERIAL LINES: SLIP. 1988. RFC 1055.

[53] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, pages 275–287.

[54] The Go Programming Language. URL http://golang.org/.

[55] Unikernel. URL http://en.wikipedia.org/wiki/Unikernel.

[56] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. ACM SIGOPS Operating Systems Review 36, pages 181–194.

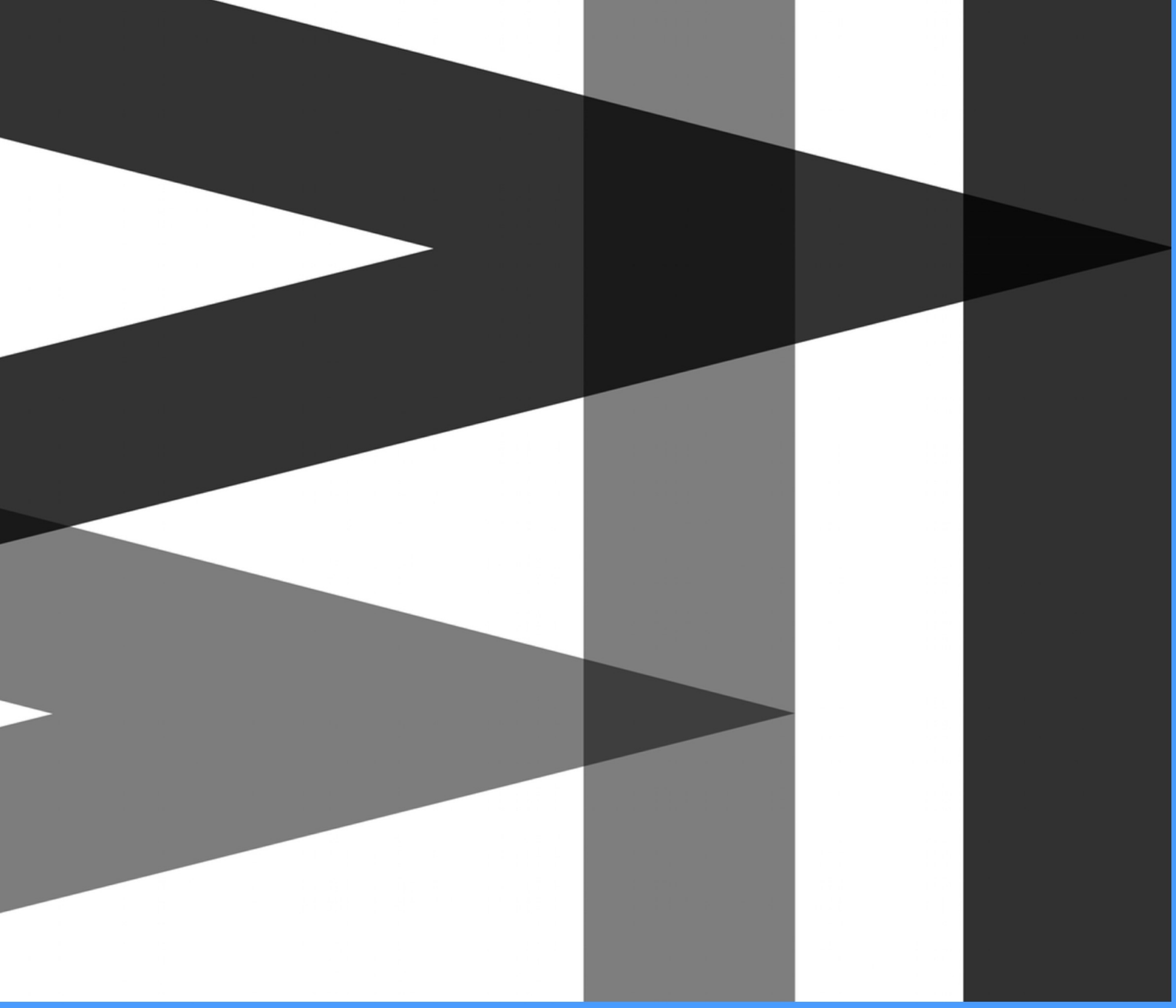
[57] Zhikui Wang, Xiaoyun Zhu, Pradeep Padala, and Sharad Singhal. May 2007. Capacity and Performance Overhead in Dynamic Resource Allocation to Virtual Containers. In: Proceedings of the IFIP/IEEE Symposium on Integrated Management, pages 149–158.

[58] Gary R. Wright and W. Richard Stevens. 1995. TCP/IP Illustrated, Volume 2. Addison Wesley. ISBN 0-201-63354-X.

[59] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. 2006. Automatically Generating Malicious Disks using Symbolic Execution. In: Proceedings of the IEEE Symposium on Security and Privacy, pages 243–257.

[60] Arnaud Ysmal and Antti Kantee. 2009. Fs-utils: File Systems Access Tools for Userland. In: Proceedings of EuroBSDCon. URL `http://www.netbsd.org/~stacktic/ebc09_fs-utils_paper.pdf`.

www.dbooks.org



Includes the Rumprun Unikernel

Rump kernels enable you to build the software stack you need without forcing you to reinvent the wheels.

The key observation is that a software stack needs driver-like components which are conventionally tightly-knit into operating systems — even if you do not desire the limitations and infrastructure overhead of a given OS, you do need drivers.