Department of Computer Science and Engineering

Flexible Operating System Internals:

The Design and Implementation of the Anykernel and Rump Kernels

Antti Kantee

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 171/2012

Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels

Antti Kantee

A doctoral dissertation completed for the degree of Doctor of Science in Technology to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T1 of the school on 7 December 2012 at 12 noon.

Aalto University
School of Science
Department of Computer Science and Engineering

Supervising professor
Professor Heikki Saikkonen

Preliminary examiners
Dr. Marshall Kirk McKusick, USA
Professor Renzo Davoli, University of Bologna, Italy

Opponent
Dr. Peter Tröger, Hasso Plattner Institute, Germany

Aalto University publication series
DOCTORAL DISSERTATIONS 171/2012

ISBN 978-952-60-4916-8 (printed)
ISBN 978-952-60-4917-5 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-60-4917-5

Unigrafia Oy
Helsinki 2012

Finland

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Antti Kantee

Name of the doctoral dissertation
Flexible Operating System Internals:
The Design and Implementation of the Anykernel and Rump Kernels

Publisher School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 171/2012

Field of research Software Systems

Manuscript submitted 21 August 2012 **Date of the defence** 7 December 2012

Permission to publish granted (date) 9 October 2012 **Language** English

☒ **Monograph** ☐ **Article dissertation (summary + original articles)**

Abstract

The monolithic kernel architecture is significant in the real world due to the large amount of working and proven code. However, the architecture is not without problems: testing and development is difficult, virtualizing kernel services can be done only by duplicating the entire kernel, and security is weak due to a single domain where all code has direct access to everything. Alternate kernel architectures such as the microkernel and exokernel have been proposed to rectify these problems with monolithic kernels. However, alternate system structures do not address the common desire of using a monolithic kernel when the abovementioned problems do not apply.

We propose a flexible anykernel architecture which enables running kernel drivers in a variety of configurations, examples of which include microkernel-style servers and application libraries. A monolithic kernel is shown to be convertible into an anykernel with a reasonable amount of effort and without introducing performance-hindering indirection layers. The original monolithic mode of operation is preserved after the anykernel adjustments, and alternate modes of operation for drivers are available as a runtime choice. For non-monolithic modes of operation, the rump kernel is introduced as a lightweight container for drivers. A rump kernel runs on top of a hypervisor which offers high-level primitives such as thread scheduling and virtual memory. A production quality implementation for the NetBSD open source OS has been done. The anykernel architecture and rump kernels are evaluated both against four years of real-world experience from daily NetBSD development as well as against synthetic benchmarks.

Keywords operating system kernel architecture, implementation, open source

ISBN (printed) 978-952-60-4916-8 **ISBN (pdf)** 978-952-60-4917-5

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2012

Pages 358 **urn** http://urn.fi/URN:ISBN:978-952-60-4917-5

Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä
Antti Kantee

Väitöskirjan nimi
Joustavat käyttöjärjestelmät: Jokaytimen ja Tynkäytimien suunnittelu ja toteutus

Julkaisija Perustieteiden korkeakoulu

Yksikkö Tietotekniikan laitos

Sarja Aalto University publication series DOCTORAL DISSERTATIONS 171/2012

Tutkimusala Ohjelmistojärjestelmät

Käsikirjoituksen pvm 21.08.2012 **Väitöspäivä** 07.12.2012

Julkaisuluvan myöntämispäivä 09.10.2012 **Kieli** Englanti

☒ **Monografia** ☐ **Yhdistelmäväitöskirja (yhteenveto-osa + erillisartikkelit)**

Tiivistelmä

Monoliittisen ytimen käyttöjärjestelmät ovat merkittäviä, koska ne sisältävät suuren määrän olemassaolevia ja testattja ajureita. Monoliittinen arkkitehtuuri ei kuitenkaan ole ongelmaton: virtualisointi onnistuu vain virtualisoimalla ydin kokonaisuutena, ytimen testaus on vaikeaa ja tietoturva on heikkoa johtuen siitä, että ytimessä suoritettavalla ohjelmistolla on suora pääsy käyttöjärjestelmän kaikkiin osiin. Vaihtoehtoisia ydinarkkitehtuureita, kuten mikroydin ja eksoydin, on ehdotettu korjaamaan monoliittisen arkkitehtuurin ongelmia. Vaihtoehtoisissa malleissa on kuitenkin ongelmana se, etteivät ne tue yleistä halua käyttää monoliittista ydintä, kun yllämainitut ongelmat eivät ole relevantteja.

Tässä työssä luodaan joustava jokaydin. Se mahdollistaa ajureiden suorittamisen monissa konfiguraatioissa, esimerkkejä joista ovat mikroydin-tyyliset palvelimet ja sovelluskirjastot. Monoliittisen ytimen muuntamisen jokaytimeksi näytetään onnistuvan säädyllisellä vaivalla ilman suorituskykyä heikentäviä abstraktioita. Mahdollisuus suorittaa ajureita monoliittisessä ytimessä säilyy ja muut suorituskonfiguraatiot tarjotaan ajonaikaisena valintana. Muita kuin monoliittisista suorituskonfiguraatiota varten määritellään tynkäydin, joka on ajoympäristö ajureille. Tynkäytimen suoritus tapahtuu hyperkaihtimen päällä. Hyperkaihdin tarjoaa tynkäytimelle korkean tason palveluita kuten säieskeduloinnin ja virtuaalimuistin. Toteutus on tehty tuotantotasoisena NetBSD-nimiselle avoimen lähdekoodin käyttöjärjestelmälle. Jokaydinarkkitehtuuri ja tynkäytimet arvioidaan sekä neljän vuoden reaalikokemuksen että synteettisten kokeiden perusteella.

Avainsanat käyttöjärjestelmäydinarkkitehtuuri, toteutus, avoin lähdekoodi

ISBN (painettu) 978-952-60-4916-8 **ISBN (pdf)** 978-952-60-4917-5

ISSN-L 1799-4934 **ISSN (painettu)** 1799-4934 **ISSN (pdf)** 1799-4942

Julkaisupaikka Espoo **Painopaikka** Helsinki **Vuosi** 2012

Sivumäärä 358 **urn** http://urn.fi/URN:ISBN:978-952-60-4917-5

To the memory of my grandfather,
Lauri Kantee

Preface

Meet the new boss
Same as the old boss
– The Who

The document you are reading is an extension of the production quality implementation I used to verify the statements that I put forth in this dissertation. The implementation and this document mutually support each other, and cross-referencing one while studying the other will allow for a more thorough understanding of this work. As usually happens, many people contributed to both, and I will do my best below to account for the ones who made major contributions.

In the professor department, my supervisor Heikki Saikkonen has realized that figuring things out properly takes a lot of time, so he did not pressure me for status updates when there was nothing new worth reporting. When he challanged me on my writing, his comments were to the point. He also saw the connecting points of my work and put me in touch with several of my later key contacts.

At the beginning of my journey, I had discussions with Juha Tuominen, the professor I worked for during my undergraduate years. Our discussions about doctoral work were on a very high level, but I found myself constantly going back to them, and I still remember them now, seven years later.

I thank my preliminary examiners, Kirk McKusick and Renzo Davoli, for the extra effort they made to fully understand my work before giving their verdicts. They not only read the manuscript, but also familiarized themselves with the implementation and manual pages.

My friend and I daresay mentor Johannes Helander influenced this work probably more than any other person. The previous statement is somewhat hard to explain, since we almost never talked specifically about my work. Yet, I am certain that the views and philosophies of Johannes run deep throughout my work.

Andre Dolenc was the first person to see the potential for using this technology in a product. He also read some of my earlier texts and tried to guide me to a "storytelling" style of writing. I think I may finally have understood his advice.

The first person to read a complete draft of the manuscript was Thomas Klausner. Those who know Thomas or his reputation will no doubt understand why I was glad when he offered to proofread and comment. I was rewarded with what seemed like an endless stream of comments, ranging from pointing out typos and grammatical mistakes to asking for clarifications and to highlighting where my discussion was flawed.

The vigilance of the NetBSD open source community kept my work honest and of high quality. I could not cut corners in the changes I made, since that would have been called out immediately. At the risk of doing injustice to some by forgetting to name them, I will list a few people from the NetBSD community who helped me throughout the years. Arnaud Ysmal was the first person to use my work for implementing a large-scale application suite. Nicolas Joly contributed numerous patches. I enjoyed the many insightful discussions with Valeriy Ushakov, and I am especially thankful to him for nudging me into the direction of a simple solution for namespace protection. Finally, the many people involved in the NetBSD test effort helped me to evaluate one of the main use cases for this work.

Since a doctoral dissertation should not extensively repeat previously published material, on the subject of contributions from my family, please refer to the preface of [Kantee 2004].

Then there is my dear Xi, who, simply put, made sure I survived. When I was hungry, I got food. When I was sleepy, I got coffee, perfectly brewed. Sometimes I got tea, but that usually happened when I had requested it instead of coffee. My output at times was monosyllabic at best ("hack", "bug", "fix", ...), but that did not bother her, perhaps because I occasionally ventured into the extended monosyllabic spectrum ("debug", "coffee", "pasta", ...). On top of basic caretaking, I received assistance when finalizing the manuscript, since I had somehow managed to put myself into a situation characterized by a distinct lack of time. For example, she proofread multiple chapters and checked all the references fixing several of my omissions and mistakes. The list goes on and on and Xi quite literally goes to 11.

This work was partially funded by the following organizations: Finnish Cultural Foundation, Research Foundation of the Helsinki University of Technology, Nokia Foundation, Ulla Tuominen Foundation and the Helsinki University of Technology. I thank these organizations for funding which allowed me to work full-time pursuing my vision. Since the time it took for me to realize the work was much longer than the time limit for receiving doctoral funding, I also see fit to extend my gratitude to organizations which have hired me over the years and thus indirectly funded this work. I am lucky to work in field where this is possible and even to some point synergetic with the research.

Lastly, I thank everyone who has ever submitted a bug report to an open source project.

Antti Kantee

November 2012

Contents

Appendix C Patches to the 5.99.48 source tree

List of Abbreviations

ABI	Application Binary Interface: The interface between two binaries. ABI-compatible binaries can interface with each other.
ASIC	Application-Specific Integrate Circuit
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
i386	Intel 32-bit ISA (a.k.a. IA-32)
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
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	Round Trip 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

List of Figures

(with 9 illustrations)

List of Tables

1 Introduction

In its classic role, an operating system is a computer program which abstracts the platform it runs on and provides services to application software. Applications in turn provide functionality that the user of the computer system is interested in. For the user to be satisfied, the operating system must therefore support both the platform and application software.

An operating system is understood to consist of the kernel, userspace libraries and utilities, although the exact division between these parts is not definitive in every operating system. The kernel, as the name says, contains the most fundamental routines of the operating system. In addition to low-level platform support and critical functionality such as thread scheduling and IPC, the kernel offers *drivers*, which abstract an underlying entity. Throughout this dissertation we will use the term *driver* in an extended sense which encompasses not only hardware device drivers, but additionally for example file system drivers and the TCP/IP network driver.

Major contemporary operating systems follow the monolithic kernel model. Of popular general purpose operating systems, for example Linux, Windows and Mac OS X are regarded as monolithic kernel operating systems. A monolithic kernel means that the entire kernel is executed in a single privileged domain, as opposed to being spread out to multiple independent domains which communicate via message passing. The single privileged domain in turn means that all code in the kernel has full capability to directly control anything running on that particular system. Furthermore, the monolithic kernel does not inherently impose any technical restrictions for the structure of the kernel: a routine may call any other routine in the kernel and access all memory directly. However, like in most disciplines, a well-designed architecture is desirable. Therefore, even a monolithic kernel tends towards structure.

1.1 Challenges with the Monolithic Kernel

Despite its widespread popularity, we identified a number of suboptimal characteristics in monolithic kernels which we consider as the motivating problems for this dissertation:

1. **Weak security and robustness**. Since all kernel code runs in the same privileged domain, a single mistake can bring the whole system down. The fragile nature of the monolithic kernel is a long-standing problem to which all monolithic kernel operating systems are vulnerable.

   Bugs are an obvious manifestation of the problem, but there are more subtle issues to consider. For instance, widely used file system drivers are vulnerable against untrusted disk images [115]. This vulnerability is acknowledged in the manual page of the mount command for example on Linux: "*It is possible for a corrupted file system to cause a crash*". The commonplace act of accessing untrusted removable media such as a USB stick or DVD disk with an in-kernel file system driver opens the entire system to a security vulnerability.

2. **Limited possibilities for code reuse**. The code in a monolithic kernel is viewed to be an all-or-nothing deal due to a belief that everything is intertwined with everything else. This belief implies that features cannot be cherry-picked and put into use in other contexts and that the kernel drivers have value only when they are a part of the monolithic kernel. Examples include file systems [115] and networking [82].

   One manifestation of this belief is the reimplementation of kernel drivers for userspace. These reimplementations include TCP/IP stacks [27, 96] and file system drivers [2, 89, 105] [1] for the purposes of research, testing, teaching

[1] We emphasize that with file systems we do **not** mean FUSE (Filesystem in Userspace) [106]. FUSE provides a mechanism for attaching a file system driver as a microkernel style server, but does not provide the driver itself. The driver attached by FUSE may be an existing kernel driver which was reimplemented in userspace [2, 3].

and application-level drivers. The common approaches for reimplementation are starting from scratch or taking a kernel driver and adjusting the code until the specific driver can run in userspace.

If cherry-picking unmodified drivers were possible, kernel drivers could be directly used at application level. Code reuse would not only save the initial implementation effort, but more importantly it would save from having to maintain the second implementation.

3. **Development and testing is convoluted**. This is a corollary of the previous point: in the general case testing involves booting up the whole operating system for each iteration. Not only is the bootup slow in itself, but when it is combined with the fact that an error may bring the entire system down, development cycles become long and batch mode regression testing is difficult. The difficulty of testing affects how much testing and quality assurance the final software product receives. Users are indirectly impacted: better testing produces a better system.

   Due to the complexity and slowness of in-kernel development, a common approach is to implement a prototype in userspace before porting the code to the kernel. For example FFS in BSD [70] and ZFS in Solaris [16] were implemented this way. This approach may bring additional work when the code is being moved into the kernel, as the support shim in userspace may not have fully emulated all kernel interfaces [70].

1.2 Researching Solutions

One option for addressing problems in monolithic kernels is designing a better model and starting from scratch. Some examples of alternative kernel models include the microkernel [9, 43, 45, 64] Exokernel [33] and a partitioned kernel [12, 112]. The problem with starting from scratch is getting to the point of having enough

support for external protocols to be a viable alternative for evaluation with real world applications. These external protocols include anything serviced by a driver and range from a networking stack to a POSIX interface. As the complexity of the operating environment and external restrictions grow, it is more and more difficult to start working on an operating system from scratch [92]. For a figure on the amount of code in a modern OS, we look at two subsystems in the Linux 3.3 kernel from March 2012. There are 1,001,218 physical lines of code for file system drivers in the `fs` subdirectory and 711,150 physical lines of code for networking drivers in the `net` subdirectory (the latter figure does not include NIC drivers, which are kept elsewhere in the source tree). For the sake of discussion, let us assume that a person who can write 100 lines of bugfree code each day writes all of those drivers. In that case, it will take over 46 years to produce the drivers in those subdirectories.

Even if there are resources to write a set of drivers from scratch, the drivers have not been tested in production in the real world when they are first put out. Studies show that new code contains the most faults [18, 91]. The faults do not exist because the code would have been poorly tested before release, but rather because it is not possible to anticipate every real world condition in a laboratory environment. We argue that real world use is the property that makes an existing driver base valuable, not just the fact that it exists.

1.3 Thesis

We claim that it is possible to construct a flexible kernel architecture which solves the challenges listed in Section 1.1, and yet retain the monolithic kernel. Furthermore, it is possible to implement the flexible kernel architecture solely by good programming principles and without introducing levels of indirection which hinder the monolithic kernel's performance characteristics. We show our claim to be true by an implementation for a BSD-derived open source monolithic kernel OS, NetBSD [87].

We define an *anykernel* to be an organization of kernel code which allows the kernel's *unmodified* drivers to be run in various configurations such as application libraries and microkernel style servers, and also as part of a monolithic kernel. This approach leaves the configuration the driver is used in to be decided at runtime. For example, if maximal performance is required, the driver can be included in a monolithic kernel, but where there is reason to suspect stability or security, the driver can still be used as an isolated, non-privileged server where problems cannot compromised the entire system.

An anykernel can be instantiated into units which virtualize the bare minimum support functionality for kernel drivers. We call these virtualized kernel instances *rump kernels* since they retain only a part of the original kernel. This minimalistic approach makes rump kernels fast to bootstrap (˜10ms) and introduces only a small memory overhead (˜1MB per instance). The implementation we present hosts rump kernels in unprivileged user processes on a POSIX host. The platform that the rump kernel is hosted on is called the host platform or *host*.

At runtime, a rump kernel can assume the role of an application library or that of a server. Programs requesting services from rump kernels are called rump kernel clients. Throughout this dissertation we use the shorthand *client* to denote rump kernel clients. We define three client types.

1. Local: the rump kernel is used in a library capacity. Like with any library, using the rump kernel as a library requires that the application is written to use APIs provided by a rump kernel. The main API for a local client is a system call API with the same call signatures as on a regular NetBSD system.

   For example, it is possible to use a kernel file system driver as a library in an application which interprets a file system image.

2. Microkernel: the host routes client requests from regular processes to drivers running in isolated servers. Unmodified application binaries can be used.

   For example, it is possible to run a block device driver as a microkernel style server, with the kernel driver outside the privileged domain.

3. Remote: the client and rump kernel are running in different containers (processes) with the client deciding which services to request from the rump kernel and which to request from the host kernel. For example, the client can use the TCP/IP networking services provided by a rump kernel. The kernel and client can exist either on the same host or on different hosts. In this model, both specifically written applications and unmodified applications can use services provided by a rump kernel. The API for specifically written applications is the same as for local clients.

   For example, it is possible to use an unmodified Firefox web browser with the TCP/IP code running in a rump kernel server.

Each configuration contributes to solving our motivating problems:

1. **Security and robustness**. When necessary, the use of a rump kernel will allow unmodified kernel drivers to be run as isolated microkernel servers while preserving the user experience. At other times the same driver code can be run in the original fashion as part of the monolithic kernel.

2. **Code reuse**. A rump kernel may be used by an application in the same fashion as any other userlevel library. A local client can call any routine inside the rump kernel.

3. **Development and testing**. The lightweight nature and safety properties of a rump kernel allow for safe testing of kernel code with iteration times in the millisecond range. The remote client model enables the creation of tests using familiar tools.

Our implementation supports rump kernels for file systems [55], networking [54] and device drivers [56]. Both synthetic benchmarks and real world data gathered from a period between 2007 and 2011 are used for the evaluation. We focus our efforts at drivers which do not depend on a physical backend being present. Out of hardware device drivers, support for USB drivers has been implemented and verified. We expect it is possible to support generic unmodified hardware device drivers in rump kernels by using previously published methods [62].

1.4 Contributions

The original contributions of this dissertation are as follows:

1. The definition of an anykernel and a rump kernel.

2. Showing that it is possible to implement the above in production quality code and maintain them in a real world monolithic kernel OS.

3. Analysis indicating that the theory is generic and can be extended to other operating systems.

1.5 Dissertation Outline

Chapter 2 defines the concept of an anykernel and explains rump kernels. Chapter 3 discusses the implementation and provides microbenchmarks as supporting evidence for implementation decisions. Chapter 4 evaluates the solution. Chapter 5 looks at related work. Chapter 6 provides concluding remarks.

1.6 Further Material

1.6.1 Source Code

The implementation discussed in this dissertation can be found in source code form from the NetBSD CVS tree as of March 31st 2011 23:59UTC.

NetBSD is an evolving open source project with hundreds of volunteers and continuous change. *Any statement we make about NetBSD reflects solely the above timestamp and no other.* It is most likely that statements will apply over a wide period of time, but it is up to the interested reader to verify if they apply to earlier or later dates.

It is possible to retrieve the source tree with the following command:

```
cvs -d anoncvs@anoncvs.netbsd.org:/cvsroot co -D'20110331 2359UTC' src
```

Whenever we refer to source file, we implicitly assume the `src` directory to be a part of the path, i.e. `sys/kern/init_main.c` means `src/sys/kern/init_main.c`.

For simplicity, the above command checks out the entire NetBSD operating system source tree instead of attempting to cherry-pick only the relevant code. The checkout will require approximately 1.1GB of disk space. Most of the code relevant to this document resides under `sys/rump`, but relevant code can be found under other paths as well, such as `tests` and `lib`.

Diffs in Appendix C detail where the above source tree differs from the discussion in this dissertation.

The project was done in small increments in the NetBSD source with almost daily changes. The commits are available for study from repository provided by the NetBSD project, e.g. via the web interface at `cvsweb.NetBSD.org`.

NetBSD Release Model

We use NetBSD release cycle terminology throughout this dissertation. The following contains an explanation of the NetBSD release model. It is a synopsis of the information located at `http://www.NetBSD.org/releases/release-map.html`.

The main development branch or *HEAD* of NetBSD is known as *NetBSD-current* or, if NetBSD is implied, simply *-current*. The source code used in this dissertation is therefore -current from the aforementioned date.

Release branches are created from -current and are known by their major number, for example NetBSD 5. Release branches get bug fixes and minor features, and releases are cut from them at suitable dates. Releases always contain one or more minor numbers, e.g. NetBSD 5.0.1. The first major branch after March 31st is NetBSD 6 and therefore the first release to potentially contain this work is NetBSD 6.0.

A -current snapshot contains a kernel API/ABI version. The version is incremented only when an interface changes. The kernel version corresponding to March 31st is 5.99.48. While this version number stands for any -current snapshot between March 9th and April 11th 2011, whenever 5.99.48 is used in this dissertation, it stands for -current at 20110331.

Code examples

This dissertation 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.

1.6.2 Manual Pages

Unix-style manual pages for interfaces described in this dissertation are available in Appendix A. The manual pages are taken verbatim from the NetBSD 5.99.48 distribution.

1.6.3 Tutorial

Appendix B contains a hands-on tutorial. It walks through various use cases where drivers are virtualized, such as encrypting a file system image using the kernel crypto driver and running applications against virtual userspace TCP/IP stacks. The tutorial uses standard applications and does not require writing code or compiling special binaries.

2 The Anykernel and Rump Kernels

As a primer for the technical discussion in this document, we consider the elements that make up a modern Unix-style operating system kernel. The following is not the only way to make a classification, but it is the most relevant one for our coming discussion.

The *CPU specific code* is on the bottom layer of the OS. 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 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. The currently executing thread can be switched either when the scheduler decides it has run too long, or when the thread itself makes a system call which requires waiting for a condition to become true before execution can be resumed. 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. In

NetBSD, both user processes and the kernel are preemptively scheduled, meaning the scheduler can decide to unschedule the currently executing thread and schedule a new one.

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 [15] 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 [58] for file systems and mbufs [114] for the TCP/IP stack.

Drivers interact with external objects such as file system images, hardware, the network, etc. After a fashion, it can be said they accomplish all the useful work an operating system does. It needs to be remembered, though, that they operate by

building on top of the entities mentioned earlier in this section. Drivers are what we are ultimately interested in utilizing, but to make them available we must deal with everything they depend on. Being able to reuse drivers means we have to provide semantically equivalent implementations of the support routines that the drivers use. The straightforward way is to run the entire kernel, but it not always the optimal approach, as we will demonstrate throughout this dissertation.

2.1 An Ultralightweight Virtual Kernel for Drivers

Virtualization of the entire operating system can done either by modifying the operating system kernel by means of paravirtualization (e.g. User-Mode Linux [26] or Xen [11]) or by virtualizing at the hardware layer so than an unmodified operating system can be run (by using e.g. QEMU [13]). From the perspective of the host, virtualization provides both multiplicity and isolation of the monolithic kernel, and can be seen as a possible solution for our security and testing challenges from Section 1.1. Using a fully virtualized OS as an application library is less straightforward, but can be done by bootstrapping a guest instance of an operating system and communicating with the guest's kernel through an application running on the guest. For example, libguestfs [4] uses this approach to access file system images safely.

Full OS virtualization is a heavyweight operation. For instance, several seconds of bootstrap delay for a fully virtualized OS [48] is too long if we wish to use virtualized kernel instances as application libraries — humans perceive delays of over 100ms [78]. While it may be possible to amortize the bootstrap delay over several invocations, managing cached instances adds complexity to the applications, especially if multiple different users want to use ones for multiple different purposes. Furthermore, a full OS consumes more machine resources, such as memory and storage and CPU, than is necessary for kernel driver virtualization. The increased resource requirement

is because a full OS provides the entire application environment, which from our perspective is overhead.

Virtualization via containers [52] provides better performance than paravirtualization [104, 110]. However, containers do not address our motivating problems. With containers, the host kernel is directly used for the purposes of all guests. In other words, kernel drivers are run in a single domain within the host. There is no isolation between kernel drivers for different guests and a single error in one of them can bring all of the guests and the host down. The lack of isolation is due to the use case that containers are targeted at: they provide a virtual application environment instead of a virtual kernel environment.

We wish to investigate a lightweight solution to maximize the performance and simplicity in our use cases. In other words, we believe in an approach which requires only the essential functionality necessary for solving a problem [59].

2.1.1 Partial Virtualization and Relegation

A key observation in our lightweight approach is that part of the supporting functionality required by drivers is readily provided by the system hosting our virtualized driver environment. For example, drivers need a memory address space to execute in; we use the one that the host provides instead of simulating a second one on top of it. Likewise, we directly use the host's threading and scheduling facilities in our virtual kernel instead of having the host schedule a virtual kernel with its own layer of scheduling. Relegating support functionality to the host avoids adding a layer of indirection and overhead. It is also the reason why we call our virtualized kernel instance a *rump kernel*: it virtualizes only a part of the original. In terms of taxonomy, we classify a rump kernel as partial paravirtualization.

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. 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 term *microkernel*, 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 will examine the implementation details of an anykernel more closely in Chapter 3 where we turn a monolithic kernel into an anykernel.

While rump kernels (i.e. guests) use features provided by the host, the difference to containers is that rump kernels themselves are not a part of the host. Instead, the host provides the necessary facilities for starting multiple rump kernel instances. These instances are not only isolated from each other, but also from the host. In POSIX terms, this means that a rump kernel has the same access rights as any other process running with the same credentials.

An example of a practical benefit resulting from relegating relates to program execution. When a fully virtualized operating system executes a program, it searches for the program from its file system namespace and runs it while the host remains oblivious to the fact that the guest ran a program. In contrast, the rump kernel and its clients are run from the host's file system namespace by the host. Since process execution is handled by the host, there is no need to configure a root file system for a rump kernel, and rump kernels can be used as long as the necessary binaries are present on the host. This also means that there is no extra maintenance burden resulting from keeping virtual machine images up-to-date. As long the host is kept up-to-date, the binaries used with rump kernels will not be out of date and potentially contain dormant security vulnerabilities [38].

Another example of a practical benefit that a rump kernel provides is core dump size. Since a rump kernel has a small memory footprint, the core dumps produced as the result of a *kernel panic* are small. Small core dumps significantly reduce disk use and restart time without having to disable core dumps completely and risk losing valuable debugging information.

A negative implication of selective virtualization is that not all parts of the kernel can be tested and isolated using this scheme. However, since a vast majority of kernel bugs are in drivers [18], our focus is on improving the state-of-the-art for them.

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.

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.

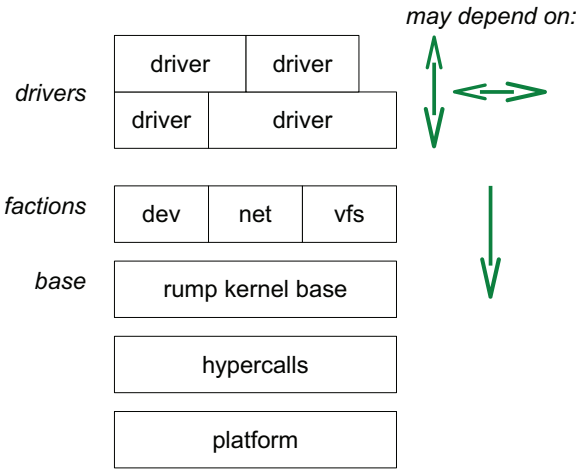

Figure 2.1: **Rump kernel hierarchy.** The desired drivers dictate the required components. The factions are orthogonal and depend only on the rump kernel base. The rump kernel base depends purely on the hypercall layer.

We 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 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) [67]. 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 applications such as network testing [44] which require thousands of virtual instances. For example, as we will later measure in Chapter 4, a virtual TCP/IP stack without file system support is 40% smaller (400kB) than one which contains file system support.

2.1.3 Hosting

A rump kernel accesses host resources through the *rumpuser* hypercall interface. The hypercall layer is currently implemented for POSIX hosts, but there is no reason why it could not be adapted to suit alternative hosting as well, such as microkernels. We analyze the requirements for the hypercall interface in more detail

in Section 3.2.3. Whenever discussing hosting and the hypercall interface we attempt to keep the discussion generic and fall back to POSIX terminology only when necessary.

The host controls what resources the guest has access to. It is unnecessary to run rump kernels as root on POSIX systems — security-conscious people will regard it as unwise as well unless there is a very good reason for it. Accessing resources such as the host file system will be limited to whatever credentials the rump kernel runs with on the host. Other resources such as memory consumption may be limited by the host as well.

2.2 Rump Kernel Clients

We define a rump kernel client to be an application that requests services from a rump kernel. Examples of rump kernel clients are an application that accesses the network through a TCP/IP stack provided by a rump kernel, or an application that reads files via a file system driver running in a rump kernel. 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.

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. The difference is that a rump kernel client must explicitly request all services from the rump kernel, while a process receives some services such as scheduling and memory protection implicitly from the host.

As mentioned in Chapter 1 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

```
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.

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.

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 VFS-bypassing example in Figure 2.3 is a local client which manipulates the kernel directly, while the local client in Figure 2.2 uses established interfaces.

  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 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

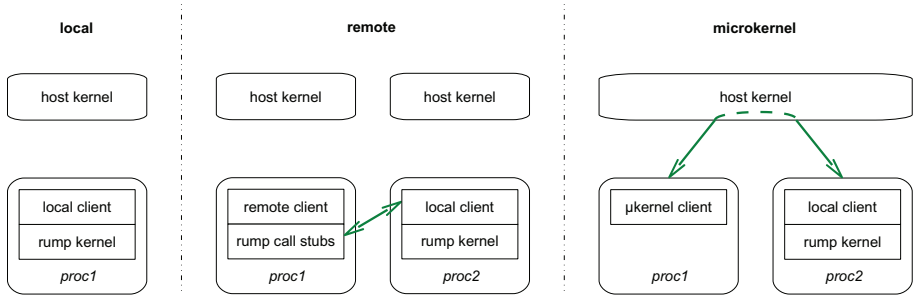

Figure 2.4: **Client types illustrated.** For local clients the client and rump kernel reside in a single process, while remote and microkernel clients reside in separate processes and therefore do not have direct memory access into the rump kernel.

concept of executing different configuration steps as multiple invocations of the tool, adaptation of the configuration code may not always be simple.

On a POSIX system, 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 on the local host or 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 [37] if it is important to have the request routing policy outside of the client.

  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.

We have implemented support for remote clients which communicate with the server using local domain sockets or TCP sockets. Using sockets is not the only option, and for example the `ptrace()` facility can also be used to implement remote clients [26, 37].

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.

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.

- **Microkernel** clients 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 [53]) and *character/block* device drivers (pud [84]). 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 process, 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 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 in 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` [2]. The soft context determines the value of `curlwp`.

[2] `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`. The pointer is cached directly in the thread structure to avoid extra indirection.

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 [10].

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* at A–23. 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, as they now need a separate call per host thread. 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.

2.3.1 Kernel threads

Up until now, we have discussed the rump kernel context of threads which are created by the client, typically by calling `pthread_create()`. 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.

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 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 and fail. 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

```
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.

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.

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 an example to clarify the content of this subsection. 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.

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 is relegated to the host because support in a rump kernel would not add value in terms of the intended use cases. The only case where full virtual memory support would be helpful would be for testing the virtual memory subsystem. 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 [11]. Therefore, supporting a corner use case was not seen worth the performance penalty in other use cases.

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

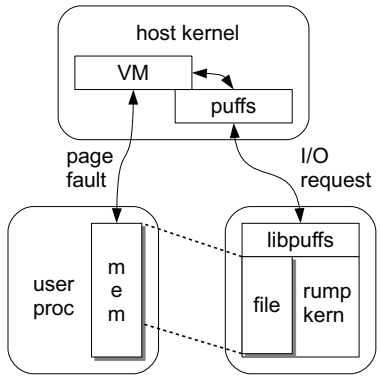

Figure 2.6: **Providing memory mapping support on top of a rump kernel**. The file is mapped into the client's address space by the host kernel. When non-resident pages in the mapped range are accessed by the client, a page fault is generated and the rump kernel is invoked via the host kernel's file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

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 *on top* of rump kernels. In fact, when running file systems as microkernel servers, the puffs [53] userspace file system framework and the host kernel provide memory mapping 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 [80] can be investigated for memory mapping support in remote clients.

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 [81]. 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 marshalls 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 [93].

2.6 Summary

A rump kernel is a partial virtualization of an operating system kernel with the virtualization target being the drivers. To be as lightweight as possible, a rump kernel relies on two features: relegating support functionality to the host where possible 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.

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 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

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 are believed to apply to other similar operating systems.

3.1 Kernel Partitioning

As mentioned in Section 2.1, 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 executable 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 an executable, 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, 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 three 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.

3. **weak symbols**. A weak symbol is one which is used in case no other definition is available. If a non-weak symbol is linked in the same set, it is used and the weak symbol is silently ignored by the linker. Essentially, weak symbols can be used to provide a default stub implementation. However, as opposed to function pointers, weak symbols are a linker concept instead of a program concept. It is not possible to deduce behavior by looking at the code at the callsite. Unlike a function pointer, using a weak symbol also prevents a program from reverting from a non-weak definition back to a weak definition. Therefore, the weak symbol technique cannot be used anywhere where code can be unloaded at runtime. We used stub implementations defined by weak symbols very sparingly because of the above reasons and preferred other approaches.

Finally, to illustrate the problems and 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 a file system has been mounted [3].

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

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

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. 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.

   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 VM, and we have extracted it.

3. **bypassed layers** such as scheduling. They need completely 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

On a POSIX system, the natural way to provide components to be linked together is with libraries. Rump kernel components are compiled as part of the regular system build and installed as libraries into `/usr/lib`. 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.

By default, NetBSD installs two production variants of each library: a static library and a shared library. The system default is to produce dynamically linked executables which use shared libraries, and this approach is also used when linking rump kernels. In custom built rump kernels the choice is up to the user. Shared libraries allow the host to load the components into physical memory only once irrespective of how many rump kernel instances are started, but shared libraries have worse performance due to indirection [39]. Figure 3.1 illustrates the speed penalty inherent to the position independent code in shared libraries by measuring the time it takes

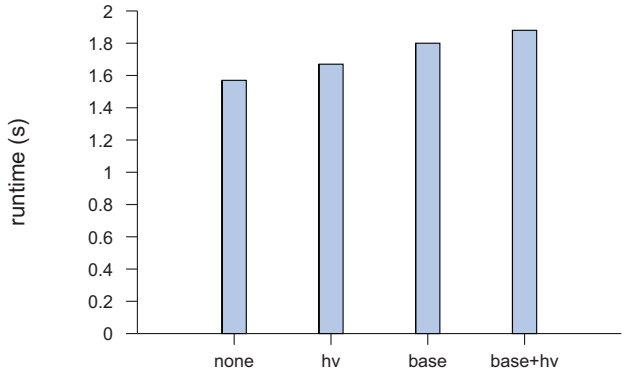

Figure 3.1: **Performance of position independent code (PIC).** A regular kernel is compiled as non-PIC code. This compilation mode is effectively the same as "none" in the graph. If the hypervisor and rump kernel base use PIC code, the execution time increases as is expected. In other words, rump kernels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

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.

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 hosting a rump kernel in a process on a POSIX host.

3.2.1 C Symbol Namespaces

In the regular case, the kernel and process 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 kernel need to be handled by the in-kernel implementation.

Single address space operating systems provide a solution [23], but require a different calling convention. On the other hand, C preprocessor macros were used by OSKit [34] to rename conflicting symbols and allow multiple different namespaces to be linked together. UML [26] 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 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.

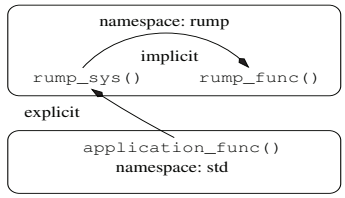

Figure 3.2: **C namespace protection**. When referencing a rump kernel symbol from outside of the rump kernel, the prefix must be explicitly included in the code. All references from inside the rump kernel implicitly contain the prefix due to bulk symbol renaming. Corollary: it is not possible to access a symbol outside the rump kernel namespace from inside the rump kernel.

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.

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.

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 2.5 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. For example, Xen [11] uses hypercalls, User Mode Linux [26] does not use privileged instructions in the usermode machine dependent code, and QEMU [13] handles such instructions in the machine emulator.

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

The hypercall library implements the hypercall interface for the particular host. On a POSIX host, the library is called `librumpuser` and the source code can be found from `lib/librumpuser`. For historical reasons, the interface itself is also called *rumpuser*, although for example *rumphyper* would be a more descriptive name.

As an example of a hypercall implementation we consider allocating "physical" memory from the host. This hypercall is implemented by the hypercall library with a call to `posix_memalign()`[4]. Conversely, the hypercall for releasing memory back to the system is implemented with a call to `free()`.

Theoretically, a rump kernel can accomplish this by calling the host directly. However, there are a number of reasons a separate hypercall interface is better.

1. The user namespace is not available in the kernel. To for example make the `posix_memalign()` call, the rump kernel has to define the prototype for the function itself. If the prototype matches what libc expects, the call is correctly resolved at link time. The downside is that duplicated information can always go out-of-date. Furthermore, calling out of the kernel is not directly possible due to the symbol renaming we presented in Section 3.2.1.

2. A separate interface helps when hosting a rump kernel on a platform which is not native to the rump kernel version. For example, while all POSIX platforms provide `stat(const char *path, struct stat *sb)`, there is no standard for the binary representation of `struct stat`. Therefore, a reference to the structure cannot be passed over the interface as binary data,

[4] `posix_memalign()` is essentially `malloc()`, but it takes an alignment parameter in addition to the size parameter. Kernel code assumes that allocating a page of memory will return it from a page-aligned offset, and using `posix_memalign()` instead of `malloc()` allows to guarantee that memory allocated by the hypercall will be page-aligned.

since the representation might not be the same in the kernel and the hypercall library. Even different versions of NetBSD have different representations, for example due to increasing `time_t` from 32bit to 64bit. Therefore, to have a well-defined interface when running on a non-native host, the hypercall interface should only use data types which are the same regardless of ABI. For example, the C99 constant width type `uint64_t` is preferred over `time_t`. Since the hypercall executes in the same process as the rump kernel, byte order is not an issue, and the native one can be used.

3. A separate interface helps future investigation of running rump kernels on non-POSIX platforms, such as microkernels. This investigation, however, will require adjustments to the hypercall interface. While the need for adjustments is foreseeable, the exact forms of the adjustments are not, and that is the reason they are not already in place.

The header file `sys/rump/include/rump/rumpuser.h` defines the hypercall interfaces. All hypercalls by convention begin with the string "rumpuser". This prevents hypercall interface references in the rump kernel from falling under the jurisdiction of symbol renaming.

We divide the hypercall interfaces into mandatory and optional ones. The routines that **must** be implemented by the rumpuser library are:

- **memory management**: allocate aligned memory, free
- **thread management**: create and join threads, TLS access
- **synchronization routines**: mutex, read/write lock, condition variable. This class is illustrated in Figure 3.3.
- **exit**: terminate the host container. In a POSIX environment depending on the parameters termination is done by calling either `abort()` or `exit()`.

```
void rumpuser_mutex_init(struct rumpuser_mtx **);
void rumpuser_mutex_init_kmutex(struct rumpuser_mtx **);
void rumpuser_mutex_enter(struct rumpuser_mtx *);
int  rumpuser_mutex_tryenter(struct rumpuser_mtx *);
void rumpuser_mutex_exit(struct rumpuser_mtx *);
void rumpuser_mutex_destroy(struct rumpuser_mtx *);
struct lwp *rumpuser_mutex_owner(struct rumpuser_mtx *);

void rumpuser_rw_init(struct rumpuser_rw **);
void rumpuser_rw_enter(struct rumpuser_rw *, int);
int  rumpuser_rw_tryenter(struct rumpuser_rw *, int);
void rumpuser_rw_exit(struct rumpuser_rw *);
void rumpuser_rw_destroy(struct rumpuser_rw *);
int  rumpuser_rw_held(struct rumpuser_rw *);
int  rumpuser_rw_rdheld(struct rumpuser_rw *);
int  rumpuser_rw_wrheld(struct rumpuser_rw *);

void rumpuser_cv_init(struct rumpuser_cv **);
void rumpuser_cv_destroy(struct rumpuser_cv *);
void rumpuser_cv_wait(struct rumpuser_cv *, struct rumpuser_mtx *);
int  rumpuser_cv_timedwait(struct rumpuser_cv *, struct rumpuser_mtx *,
                           int64_t, int64_t);
void rumpuser_cv_signal(struct rumpuser_cv *);
void rumpuser_cv_broadcast(struct rumpuser_cv *);
int  rumpuser_cv_has_waiters(struct rumpuser_cv *);
```

Figure 3.3: **Hypercall locking interfaces**. These interfaces use opaque lock data types to map NetBSD kernel locking operations to hypervisor operations.

Strictly speaking, the necessity of thread support and locking depends on the drivers being executed. We offer the following anecdote and discussion. At one point when working on rump kernel support, support for *gdb* in NetBSD was broken so that threaded programs could not be single-stepped (the problem has since been fixed). As a way to work around the problem, the variable `RUMP_THREADS` was created. If it is set to `0`, the rump kernel silently ignores kernel thread creation requests. Despite the lack of threads, for example file system drivers still function, because they do not directly depend on worker threads. The ability to run without threads allowed attaching the broken debugger to rump kernels and single-stepping them.

The following rumpuser interfaces called from a rump kernel can provide added functionality. Their implementation is optional.

- **host file access and I/O**: open, read/write. Host I/O is necessary if any host resources are to be accessed via files. Examples include accessing a file containing a file system image or accessing a host device via `/dev`.

- **I/O multiplexing**: strictly speaking, multiplexing operations such as `poll()` can be handled with threads and synchronous I/O operations. However, it is often more convenient to multiplex, and additionally it has been useful in working around at least one host system bug[5].

- **scatter-gather I/O**: can be used for optimizing the virtual network interface to transmit data directly from *mbufs*[114].

- **symbol management and external linking**: this is akin to the task of a bootloader/firmware in a regular system, and is required only if dynamically extending the rump kernel is desired (see Section 3.8.1).

- **host networking stack access**: this is necessary for the *sockin* protocol module (Section 3.9.1).

- **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.

[5] rev 1.11 of `sys/rump/net/lib/libvirtif/if_virt.c`.

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 3.4 and available as real code in `sys/rump/librump/rumpkern/scheduler.c`. We will discuss obtaining the thread context first.

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()`[6] 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, it is necessary to 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 may be used only by a single consumer at a time, and 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: 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 to the code.

[6] `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 3.4: **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.

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 3.5 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 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 they see in the

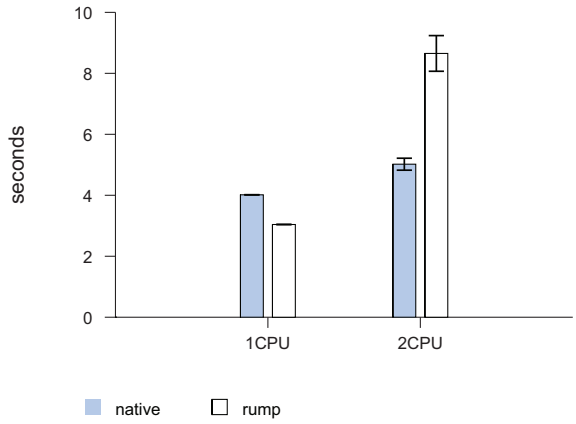

Figure 3.5: **System call performance using the trivial CPU scheduler.** While a system call into the rump kernel is faster in a single-threaded process, it is both jittery and slow for a multithreaded process. This deficiency is something we address with the advanced rump kernel CPU scheduler presented later.

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 3.6. 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.

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

Releasing a CPU requires the following steps. The pseudocode is presented in Figure 3.7. 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 3.7: **CPU release algorithm in pseudocode.** See the text for a detailed description.

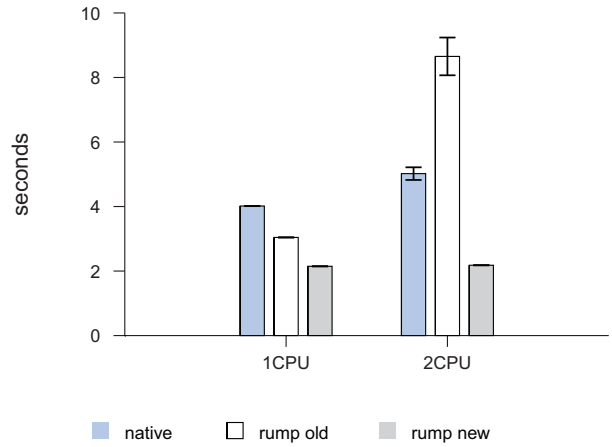

Figure 3.8: **System call performance using the improved CPU scheduler.** The advanced rump kernel CPU scheduler is lockless and cache conscious. With it, simultaneous system calls from multiple threads are over twice as fast as against the host kernel and over four times as fast as with the old scheduler.

Performance

The impact of the improved CPU scheduling algorithm is shown in Figure 3.8. 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*.

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 hardware interrupts as host threads which schedule a rump kernel CPU like other consumers, 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, but should not sleep. This property is a corollary from the previous property.
3. Softint handlers must run on the same CPU they were scheduled on. This property relaxes cross-CPU cache effects and locking requirements.
4. A softint may run only after the hardware interrupt has completed execution. 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 essentially software interrupts due to them being scheduled, a fair amount of code in NetBSD assumes that softints are supported. For example, the callout framework [19] 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 delivered until the next incoming packet was handled.

Soft interrupts are implemented in `sys/rump/librump/rumpkern/intr.c` according to the principles we listed earlier. The standard NetBSD implementation was not usable in a rump kernel since that implementation is based on direct interaction with the NetBSD scheduler.

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 [20]. While the memory 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, the standard way the kernel reads data from a file system is to memory map the file, access the mapped range, and possibly fault in missing data [101].

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[7]. Since the rump kernel does not interface with memory management hardware, 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.

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

[7] 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.

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

A regular NetBSD kernel has the ability to interact with the MMU and map physical pages to virtual addresses [8]. Since a rump kernel allocates only anonymous memory from the host, it cannot ask the host to remap any of its allocations at least on a POSIX system — there is no interface for mapping anonymous memory in multiple places. Usermode operating systems typically use a memory mapped file to represent the physical memory of the virtual kernel [26, 31]. 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 [31].

Using a file-backed solution is in conflict with the lightweight fundamentals of rump kernels. First, the file must be created at a path specified either by a configuration option or by a system guess. Furthermore, it disrupts the dynamic memory principle we have laid out. While it is relatively simple to extend the file to create more memory on demand, it is difficult to truncate the file in case the memory is no longer needed because free pages will not automatically reside in a contiguous range at the end of the file. Finally, the approach requires virtual memory mapping support from the host and limits the environments a rump kernel can be theoretically hosted

[8] NetBSD itself does not run on hardware without a MMU.

in. Therefore, we should critically examine if memory mapping capability is really needed in a rump kernel.

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 [101]. 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. 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 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 and handle the vnode pager using `mmap()` and `munmap()`.

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.

To evaluate the first option against third option, 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 3.9. We conclude that not only is copying the technologically preferred option since it avoids the need for memory mapping support on the host, it is also several times faster than `mmap()`/`munmap()` pairs.

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.

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.

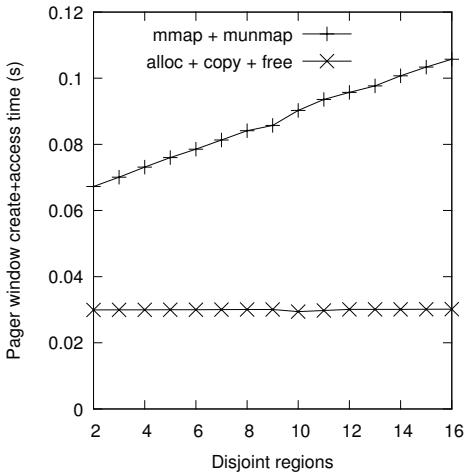

Figure 3.9: **Performance of page remapping vs. copying.** Allocating a pager window from anonymous memory and copying file pages to it for the purpose of pageout by the vnode pager is faster than remapping memory backed by a file. Additionally, the cost of copying is practically independent of the amount of non-contiguous pages. With remapping, each disjoint region requires a separate call to `mmap()`.

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.

An exception is the 4.3BSD/4.4BSD `malloc()` [73]; we stress that in this section *malloc* refers to the in-kernel implementation of the allocator. In contrast to the dynamic nature of the abovementioned allocators, the malloc implementation uses a single contiguous memory range for all allocations (`kmem_map`). The size of this range is calculated from total system memory and the range is allocated at system bootstrap time. In a rump kernel this behavior is not desirable, since it sets static limits for the system. Due to this reason we did not extract malloc and instead reimplemented the interface by relegating requests to the hypercall layer. Since malloc is being retired in NetBSD in favor of the newer style allocators, we did not see it necessary to spend effort on being able to extract the implementation instead of using the hypervisor's memory allocator directly. Notably, the situation with malloc may need to be revisited if rump kernels are to be ported to a host which does not provide a general purpose memory allocator in itself and malloc is not fully retired by then.

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. The 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 available from the host 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.

A rump kernel can be configured with either a specified amount of memory or an unlimited amount of memory. In the former case the rump kernel will disallow runtime memory allocations going over the limit. When 90% of the allocation limit has been reached, the pagedaemon will be invoked to push out memory content which has not been used recently. If no limit is specified, a rump kernel can grow up to potentially consuming all memory and swap space on the host. The latter is the default, since in our experience it suits all but a few use cases. For example, in virtually all unit test use cases the memory limit has no consequence, since a short-lived rump kernel will not have a chance to grow large enough for it to matter. Furthermore, a POSIX per-process memory size limit (*rlimit*) will usually be hit before one process can have a serious impact on the host. Hitting this limit causes the memory allocation hypercall to fail. If the memory allocation hypercall fails, the rump kernel treats the situation the same as reaching the soft-configured limit and invokes the pagedaemon, which will hopefully flush out the cache and allow the current allocation to succeed.

To free memory, the pagedaemon must locate memory resources which are not in use and release them. 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 [99] via the 4.4BSD VM. A pager has the capability to move the contents of the page in and out of secondary storage. NetBSD 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

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.

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 is critical to note that the *host* can page memory which is wired in the rump kernel barring precautions such as a `mlock()` hypercall. Most wired memory in NetBSD is in one form or another allocated from a pool as was described in Section 3.4.2. 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`. As mentioned earlier, it is invoked whenever 90% of the memory limit has been allocated from the host, or when a 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 smaller causes the pagedaemon to clear out cached memory until the smaller limit is satisfied. In contrast to the ballooning technique [109], 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.
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 sleeps only for one nanosecond. This short sleep 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 [66]. 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.

The latter option is desirable since it saves resources. However, no such standard interface exists on a standard POSIX host. The closest option is to suspend for an arbitrary period (yield, sleep, etc.). Instead, we observe that the NetBSD kernel and pthread synchronization primitives are much alike. We define a set of hypercall interfaces which provide the mutex, read/write lock and condition variable primitives. Where the interfaces do not map 1:1, such as with the `ltsleep()` and `msleep()` interfaces, we use the hypercall interfaces to emulate them (`sys/rump/librump/rumpkern/ltsleep.c`).

As usual, for a blocking hypercall we need to unschedule and reschedule the rump virtual CPU. 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. For mutexes and read/write locks we do not know a priori if we are going to block. However, 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 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 [42] 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 platform 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 [68]. 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.

Despite the lack of a locking primitive which is implemented using passive synchronization, passive synchronization is still used in the NetBSD kernel. One example which was present in NetBSD 5.99.48 is the dynamic 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`.

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.

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`.

```
/*
 * 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 3.10: **Using CPU cross calls when checking for syscall users.**

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 3.10, 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.
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 [25] (a more easily obtained description is available in "Paper 3" here [24]). 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 is the time after a thread has run on all CPUs. In the syscall example this was accomplished by running the CPU crosscall (Figure 3.10). Therefore, no modification is required as opposed to what is required for pure userspace applications to support the quiescence based RCU userspace approach [25].

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` [9]. This implementation can be used only in rump kernels with a single virtual CPU. As explained above, this implementation does not call the host pthread routines 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 3.11. Next, we analyze the results.

[9] "up" stands for uniprocessor.

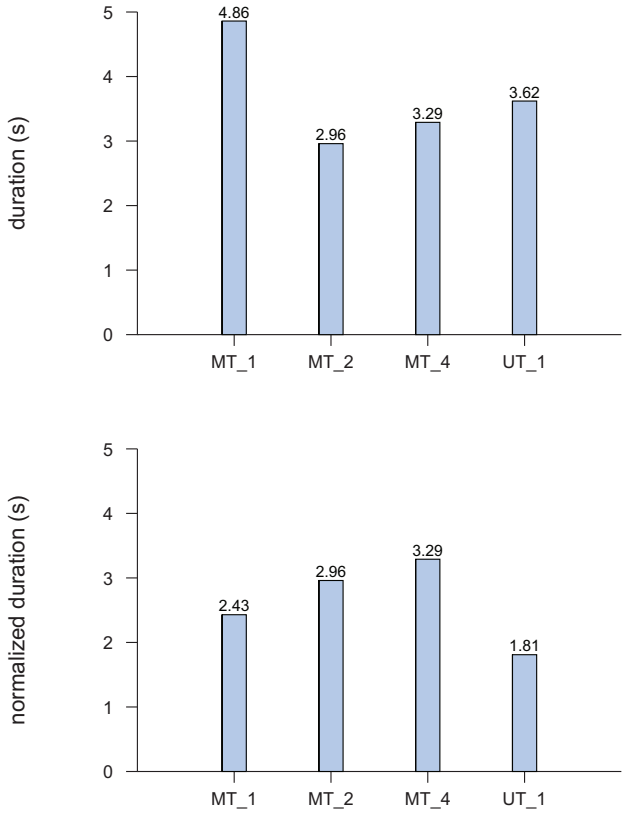

Figure 3.11: **Cost of atomic memory bus locks on a twin core host.** The first figure presents the raw measurements and the second figure presents the normalized durations per physical processor. *MT* means a multiprocessor rump kernel with hardware atomic locks and *UT* designates a uniprocessor rump 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 [12].

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

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 is

```
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 3.12: **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.

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 3.12. 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.

```
[ .... ]
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 3.13: **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` [10] 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 3.13), so the choice is a matter of taste instead of runtime performance.

[10] `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.

```
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 3.14: **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 3.14. 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 3.15: 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* at A–23). 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 3.15. 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

`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* at A–20) 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.

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.

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. On a regular system the mapping is done using the `vnconfig` utility to map regular files to `/dev/vndxx` block device nodes which can be mounted [11]. Avoiding `vnconfig` on the host is beneficial since using it requires root privileges regardless of the permissions of the backing file. With the etfs interface, a rump kernel requires only the minimal host privileges which allow it to read or write the backing file and therefore more finegrained access control is possible.

In addition to mapping files, it is 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 mod-

[11] For disk images with a partition table `vnconfig` provides multiple block devices in `/dev`. The `withsize` variant of the etfs interface can be used to map a range of the host file corresponding to the desired partition. The *p2k* and *ukfs* libraries' interfaces for mounting disk file systems provide support for this variant (manual pages at A–12 and A–38, respectively).

ule 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.

3.8 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 adapt the terminology for our discussion.

The infrastructure for supporting modular drivers on NetBSD has been available since NetBSD 5.0 [12]. Some drivers offered by NetBSD are modular, and others are

[12] 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

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.

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.

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.

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

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.

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 of the hosting process: the bootstrapping of the rump kernel by calling `rump_init()`. Both the linking of the rump kernel binary with `cc ... -lrumplibs` and loading libraries with `dlopen()` before bootstrap are equivalent operations from our perspective. Any set of components including the factions and the base may be added using the above methods, provided that driver dependencies are satisfied. After the rump kernel has been bootstrapped only modular drivers may be loaded. This limitation exists because after bootstrap only drivers which have explicit support for it can be trusted to correctly plug themselves into a running kernel.

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 and places the structure into a special `.rodata` section called `modules`. Consulting this section allows the loading party to locate the init routine and finish initialization. The section is found at runtime by locating the `__start_section` and `__end_section` symbols that are generated by the linker at the starting and ending addresses of the section.

A regular kernel uses static linking. When a regular kernel is linked, all `modules` sections are gathered into one in the resulting binary. The kernel consults the contents during bootstrap to locate builtin modules and proceeds to initialize them.

The bootstrap procedure iterates over all of the `modinfo` structures. This approach is not directly applicable to a dynamically linked environment because the dynamic linker cannot generate a combined section like static linking can — consider e.g. how loading and unloading would affect the section contents at runtime. Still we, need a mechanism for locating driver init routines when the rump kernel is bootstrapped. This task of locating the init routines is done by the `rumpuser_dl_bootstrap()` hypercall. It iterates over all the dynamic shared objects present and gathers the contents of all the `modules` sections individually.

A related problem we discovered was the GNU linker changing its behavior in generating the `__start_section` and `__end_section` symbols. Older versions generated them unconditionally. In newer versions they are generated only if they are referenced at link time (the "*PROVIDE*" mechanism). Since individual components do not use these symbols, they do not get generated when the components are linked as shared libraries. To address this regression, we created the component linker script `sys/rump/ldscript.rump` which forces the generation of the symbols in shared library components and restores desired behavior.

3.8.2 Modules: Loading and Linking

Runtime loading and linking in a rump kernel includes two distinct scenarios. First, shared objects can be loaded and linked by the host's dynamic linker. Second, static objects can be loaded and linked by code in the rump kernel. We will discuss both approaches next.

Host Dynamic Linker

Using the host's dynamic linker requires that the component is in a format that the dynamic linker can understand. In practice, this requirement means that the component needs to be a shared library (ELF type `ET_DYN`). All loading and linking is done by the host, but the init routine must be run by the rump kernel. Revisiting Table 3.3, a module loaded and linked by the host is a builtin module from the perspective of the kernel.

Since there is no system call for informing the kernel about new builtin modules appearing at runtime, we defined a rump kernel interface for running the init routine. The `rump_pub_module_init()` routine takes as an argument the pointer to the beginning of the modules section and the number of `struct modinfo` pointers. An example of client side code (adapted from `lib/libukfs/ukfs.c`) is presented in Figure 3.16.

In the monolithic kernel builtin modules cannot be unloaded because they are loaded as part of the kernel at system bootstrap time, and are not placed in separately allocated memory. However, if builtin modules are loaded into a rump kernel with `dlopen()`, it is possible to unload modules which are not busy via `dlclose()`. First, the module's fini routine should be run so that other kernel subsystems are no longer aware of it. For running the fini routine to be possible, we added the concept of disabling a builtin module to NetBSD's kernel module support. What disabling does is run the fini routine of the module thereby removing the functionality from the kernel. This concept is useful also outside the scope of rump kernels, and can for example be used to disable a driver with a newly discovered security vulnerability, perhaps making it possible to postpone the system upgrade until a suitable time. A rump kernel client can disable a builtin module by calling `rump_pub_module_fini()`. If disabling is successful, it can proceed to call `dlclose()` to unload the module from memory.

```
void *handle;
const struct modinfo *const *mi_start, *const *mi_end;
int error;

if ((handle = dlopen(fname, RTLD_LAZY|RTLD_GLOBAL)) == NULL) {
        /* error branch */
}

mi_start = dlsym(handle, "__start_link_set_modules");
mi_end = dlsym(handle, "__stop_link_set_modules");
if (mi_start && mi_end) {
        error = rump_pub_module_init(mi_start,
            (size_t)(mi_end-mi_start));
        /* ... */
} else {
        /* error: not a module */
}
```

Figure 3.16: **Loading kernel modules with `dlopen()`.** Loading is a two-part process. First, the module is loaded and linked using a call to the host's `dlopen()` routine. Second, the rump kernel is informed about the newly loaded module.

We stated that linking libraries with "`cc -o ... -lrumpfoo`" and loading them with `dlopen()` before calling `rump_init()` are equivalent. One detail deserves discussion: weak symbols. By default, symbols in the text segment are resolved lazily, which means they are unresolved until they are first used. To put it another way, the dynamic linker resolves a reference to `afunction()` only when, or rather if, it is called at runtime. However, lazy resolution does not apply to variables, including function pointers, which are resolved immediately when the program is initially linked. Any function pointer with the initial value pointing to a weak stub will retain the same value even if a strong version of the same symbol is later loaded with `dlopen()`.

Function pointers must be dynamically adjusted when the rump kernel is bootstrapped instead of relying on weak symbols. In a rump kernel, the system call vector was the main user of function pointers to weak symbols (the only other place

```
for (i = 0; i < SYS_NSYSENT; i++) {
        void *sym;

        if (rump_sysent[i].sy_flags & SYCALL_NOSYS ||
            *syscallnames[i] == '#' ||
            rump_sysent[i].sy_call == sys_nomodule)
                continue;

        if ((sym = rumpuser_dl_globalsym(syscallnames[i])) != NULL) {
                rump_sysent[i].sy_call = sym;
        }
}
```

Figure 3.17: **Adjusting the system call vector during rump kernel bootstrap.** This adjustment makes sure that all entries in the system call vector includes any system calls from all components which were loaded with `dlopen()` before the rump kernel was bootstrapped.

is old-style networking soft interrupt handlers). The bootstrap time adjustment of the system call vector from `sys/rump/librump/rumpkern/rump.c` is presented in Figure 3.17 (the code presented in the figure is slightly adjusted for presentation purposes).

Since any module loaded after bootstrap must be a NetBSD kernel module, the weak symbol problem does not exist after bootstrap. Kernel modules are not allowed to rely on weak symbols. Each kernel module's init routine takes care of adjusting the necessary pointer tables, for example the syscall vector in case the loaded modules provide system calls.

The NetBSD Kernel Linker

Using the NetBSD kernel linker means letting the code in `sys/kern/subr_kobj.c` handle linking. This linker is included as part of the base of a rump kernel. As

opposed to the host's dynamic linker, the in-kernel linker supports only relocatable objects (ELF type `ET_REL`) and does not support dynamic objects (ELF type `ET_DYN`).

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 satisfy 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 a statically linked [rump] kernel we know the symbol addresses already when the executable is linked. In a dynamically linked rump kernel, the exported addresses are known only at runtime after the dynamic linker has loaded them — recall: shared libraries are position independent code and can be loaded at any address. During bootstrap, the symbol addresses are queried and loaded as part of the `rumpuser_dl_bootstrap()` hypercall. Afterwards, the kernel linker maintains the symbol table when it links or unlinks modules.

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 or by the kernel when it autoloads a module to transparently provide support for a requested service.

3.8.3 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.

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[13] 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.

[13] 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 3.18: **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.

The interface specification explicitly allows some parts of the interface to be implemented as macros. Figure 3.18 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`).

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 [65]. 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.

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 3.19: **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.

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. An example of what we consider problematic is SPARC64's definition of `curlwp`, which we previously illustrated in Figure 3.19. 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.

Autoloading

The NetBSD kernel supports autoloading of modules for extending kernel functionality transparently. To give an example, if mounting of an unsupported file system type is requested, the kernel will attempt to load a kernel module containing the appropriate driver.

Since kernel modules are loaded from a file system, loading can work only in rump kernels with the VFS faction. Second, the modules must be present in the rump kernel file system namespace. By default, architectures with binary kernel module support map in the modules directory from the host using *etfs* (Section 3.7.1). For example, an i386 rump kernel maps in `/stand/i386/5.99.48/modules`. If modules are available in the rump kernel file system namespace, autoloading works like on a regular kernel.

Notably, autoloading does not depend on using modules which are mapped from the host's file system. However, mapping avoids the extra step of having to make the module binaries visible in the rump kernel's file system namespace either via mounting or run-time copying, and is therefore a good choice for the default mode of operation.

3.8.4 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 `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 implementation places a descriptor structure in a `.rodata` section called `rump_components`.

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

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_KERN_VFS	base initialization which is done after the VFS faction has attached, e.g. base components which do VFS operations

Table 3.4: **Rump 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.

kernel code imposes. For example, the networking domains must be attached before interfaces can be configured. It is legal (and sometimes necessary) for components to define several init routines with different configuration times. We found it necessary to define eight different levels. They are presented 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.

```
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 3.20: **Example: selected contents of `component.c` for *netinet*.** 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).

An example of a component file is presented in Figure 3.20. 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/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. The vast majority of access is done via the rumpuser hypercall interface first by opening a host device and then performing I/O via read/write. In this section we will discuss the implementation possibilities and choices for the backends for networking and file systems (block devices).

3.9.1 Networking

The canonical way an operating system accesses the network is via an interface driver which communicates with a network interface device. The interface driver is at bottom of the network stack and is invoked when all layers of network processing in the OS have been performed. The interface itself has the capability for sending raw networking packets which are handed to it by the OS. Sending and receiving raw network data is regarded as a privileged operation and is not allowed for all users. Rather, unprivileged programs only have 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. We list them below and then proceed to discuss their implementations in more detail. Figure 3.21 contains an illustration.

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 testing 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. 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.

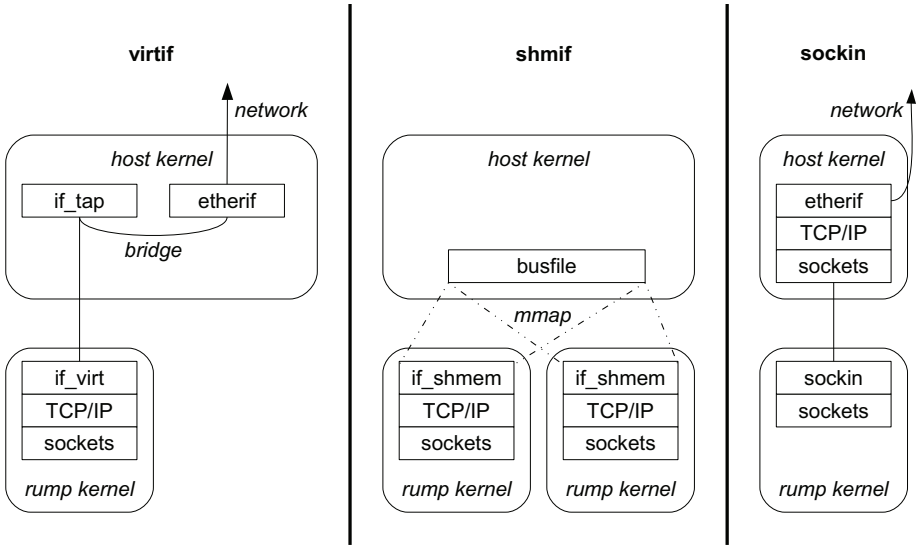

Figure 3.21: **Networking options for rump kernels.** The *virtif* facility provides a full networking stack by interfacing with the host's *tap* driver. The *shmem* facility uses interprocess shared memory to provide an Ethernet-like bus to communicate between multiple rump kernels on a single host 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.

Raw network access

The canonical way to access 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 3.21.

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

Figure 3.22: **Bridging a tap interface to the host's `re0`.** The allows the tap device to send and receive network packets via `re0`.

The commands for bridging a tap interface on NetBSD are provided in Figure 3.22. Note that IP addresses are not configured for the tap interface on the host.

The *virt* (manual page *virt.4* at A–49) 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`.

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 other nodes may be reached via routers. Accessing the host's network is possible if one rump kernel in the network uses the virt driver and routes traffic between the rump-only network and the real network.

One option for implementing packet distribution is to use a userland daemon which listens on a local domain socket [26]. 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 acts as the bus handle — all network interfaces configured to be on the same bus use the same filename. The *shmif* driver (manual page *shmif.4* at A–47) 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. However, we have never encountered this situation, so we do not consider it a serious flaw.

Sending a packet requires locking the bus and copying the contents of the packet to the buffer. Receiving packets is based on the ability of the *kqueue* interface to be able to deliver notifications to processes when a file changes. The interface driver gets a kqueue notification, it 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* at A–10) 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 host's network

Some POSIX-hosted virtualization solutions such as QEMU and UML provide unprivileged zero-configuration network access via a facility called Slirp [6]. Slirp is a program which was popular during the dial-up era. It enables running a SLIP [102] 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 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. 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 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 [114]. We call this new implementation *sockin* to reflect it being **sock**et **in**et. The attachment to the kernel is illustrated in Figure 3.23. 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. 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. The request must be handled in a timely manner, since during the period that the disk driver is handling the request

```
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 3.23: ***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.

the object the data belongs to (e.g. vm page) is held locked. 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. An unbuffered backend will 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 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.

- **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. The disk driver question is revisited in Section 4.6.5, where we compare rump kernel file system performance against an in-kernel mount.

3.10 Hardware Device Drivers: A Case of USB

Hardware device drivers require access to their counterpart to function. This counterpart can either be hardware itself or software which emulates the hardware. Throughout this section our default assumption is that the virtualized driver is controlling real hardware. Hardware access means that the hardware must be present on the system where the driver is run and that the driver has to have the appropriate privileges to access hardware. Access is typically available only when the CPU is operating in privileged (kernel) mode.

The kernel USB driver stack exports 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. While hardware must still be present, 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. 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.

Our main goal with USB support was to show it is possible to do kernel hardware device driver development in userspace. The two subproblems we had to solve were being able to attach drivers to the ugen device nodes and to integrate with the device autoconfiguration subsystem [108].

3.10.1 Structure of USB

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. Figure 3.24 contains a "dmesg" of a server with four ugenhc devices configured and one USB mass media attached.

3.10.2 Defining Device Relations with Config

Device autoconfiguration [108] is used to attach hardware device drivers in the NetBSD kernel. A configuration file determines the relationship of device drivers in the system and the autoconfiguration subsystem attempts to attach drivers according to the configuration. The configuration is expressed in a domain specific language (DSL). The language is divided into two parts: a global set of descriptions for which drivers can attach to which busses, and a system-specific configuration of what hardware is expected to be present and how this particular configuration allows devices to attach. For example, even though the USB bus allows a USB hub to be attached to another hub, the device configuration might allow a USB hub to be attached only to the host controller root hub and not other hubs.

```
golem> rump_server -lrumpvfs -lrumpdev -lrumpdev_disk -lrumpdev_usb \
       -lrumpdev_ugenhc -lrumpdev_scsipi -lrumpdev_umass -v unix:///tmp/usbserv
Copyright (c) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005,
    2006, 2007, 2008, 2009, 2010, 2011
    The NetBSD Foundation, Inc.  All rights reserved.
Copyright (c) 1982, 1986, 1989, 1991, 1993
    The Regents of the University of California.  All rights reserved.

NetBSD 5.99.48 (RUMP-ROAST) #0: Mon May  9 21:55:18 CEST 2011
        pooka@pain-rustique.localhost:/usr/allsrc/cleansrc/sys/rump/librump/rumpkern
total memory = unlimited (host limit)
timecounter: Timecounters tick every 10.000 msec
timecounter: Timecounter "rumpclk" frequency 100 Hz quality 0
cpu0 at thinair0: rump virtual cpu
cpu1 at thinair0: rump virtual cpu
root file system type: rumpfs
mainbus0 (root)
ugenhc0 at mainbus0
usb0 at ugenhc0: USB revision 2.0
uhub0 at usb0: vendor 0x7275 product 0x6d70, class 9/0, rev 0.00/0.00, addr 1
uhub0: 1 port with 1 removable, self powered
ugenhc1 at mainbus0
usb1 at ugenhc1: USB revision 2.0
uhub1 at usb1: vendor 0x7275 product 0x6d70, class 9/0, rev 0.00/0.00, addr 1
uhub1: 1 port with 1 removable, self powered
ugenhc2 at mainbus0
usb2 at ugenhc2: USB revision 2.0
uhub2 at usb2: vendor 0x7275 product 0x6d70, class 9/0, rev 0.00/0.00, addr 1
uhub2: 1 port with 1 removable, self powered
ugenhc3 at mainbus0
usb3 at ugenhc3: USB revision 2.0
uhub3 at usb3: vendor 0x7275 product 0x6d70, class 9/0, rev 0.00/0.00, addr 1
uhub3: 1 port with 1 removable, self powered
Chicony Electronics Chicony Electronics, class 0/0, rev 2.00/1.00, addr 2, uhub0 port 1 not configured
AuthenTec AuthenTec, class 255/255, rev 1.10/6.21, addr 2, uhub1 port 1 not configured
umass0 at uhub2 port 1 configuration 1 interface 0
umass0: Apple Computer product 0x1301, rev 2.00/1.00, addr 2
umass0: using SCSI over Bulk-Only
scsibus0 at umass0: 2 targets, 1 lun per target
sd0 at scsibus0 target 0 lun 0: <Apple, iPod, 2.70> disk removable
sd0: 968 MB, 30 cyl, 255 head, 63 sec, 2048 bytes/sect x 495616 sectors
golem>
```

Figure 3.24: **dmesg of USB pass-through rump kernel with mass media attached.** The mass media is probed from the host's USB ports.

The kernel device configuration language and tools were originally designed for a monolithic build. In a monolithic build one configuration is written for the entire system. This configuration is translated into a set of C language constructs by a tool called `config`, and the result is compiled into a kernel. Compiling the device information statically into the kernel is not a suitable approach for kernel modules (which we discussed in Section 3.8.1), since it is not possible to know at kernel compile time which modules may be loaded at runtime.

Since the config tool lacked support for specifying the attachment relationships of individual drivers, kernel modules typically included hand-edited tables for the specifications. Not only was creating them taxing work, but handcoding the relationships in C lacked the sanity checks performed by the config utility. Furthermore, debugging any potential errors was very cumbersome. While in the regular kernel most drivers could be compiled in at kernel build time, a rump kernel has no built-in drivers and therefore all drivers required the above style handcoding.

To address the issue, we added two new keywords to the config DSL. We will first describe them and then proceed to present examples.

- The **ioconf** keyword instructs config to generate a runtime loadable device configuration description. The created tables are loaded by the driver init routine and subsequently the autoconfiguration subsystem is invoked to probe hardware.

- The **pseudo-root** keyword specifies a local root for the configuration file. By default, config will refuse a device configuration where a device attempts to attach to an unknown parent. The pseudo-root keyword can be used to specify a parent which is assumed to be present. If the parent is not present at runtime, the driver will not attach. Without going into too much detail, pseudo-roots are usually given in the most generic form possible, e.g.

the audio driver attaches to an audiobus *interface attribute*. This generic specification means that even though not all drivers which can provide audio are known at compile time, the audio driver will still attach as long as the parent provides the correct interface attribute.

The example in Figure 3.25 specifies a USB mass media controller which may attach to a USB hub. The hub may be the root hub provided by the ugenhc host controller. The umass driver itself does not provide access to USB mass storage devices. This access is handled by disk driver and CD/DVD driver. A configuration file specifying their attachments is presented in Figure 3.26. The pseudo-root specified in this configuration file also encompasses umass, since the umass bus provides both the SCSI and ATAPI interface attributes.

It needs to be noted, though, that this example is not maximally modular: it includes configuration information both for SCSI and ATAPI disk and CD devices. This lack of modularity is due to the umass driver using old-style compile-time definitions (cf. our discussion from Section 3.1). A configuration time macro signals the presence of SCSI or ATAPI to the drivers (e.g. `NATAPIBUS`) and if one fails to be present the resulting kernel fails to link. Therefore, we include both. An improved and fully modular version of the driver would not have this compile-time limitation.

3.10.3 DMA and USB

Direct memory access (DMA) allows devices to be programmed to access memory directly without involving the CPU. Being able to freely program the DMA controller to read or write any physical memory address from an application is a security and stability issue. Allowing any unprivileged process to perform DMA on any memory address cannot be allowed in a secure and safe environment. Limiting the capability

```
#       $NetBSD: UMASS.ioconf,v 1.4 2010/08/23 20:49:53 pooka Exp $
#

ioconf umass

include "conf/files"
include "dev/usb/files.usb"

pseudo-root uhub*

# USB Mass Storage
umass*  at uhub? port ? configuration ? interface ?
```

Figure 3.25: **USB mass storage configuration.** A USB hub *pseudo-root* is used to specify a node to which umass can attach.

```
#       $NetBSD: SCSIPI.ioconf,v 1.1 2010/08/23 20:49:53 pooka Exp $
#

ioconf scsipi

include "conf/files"
include "dev/scsipi/files.scsipi"

pseudo-root scsi*
pseudo-root atapi*

# SCSI support
scsibus* at scsi?
sd*     at scsibus? target ? lun ?
cd*     at scsibus? target ? lun ?

# ATAPI support
atapibus* at atapi?
sd*     at atapibus? drive ? flags 0x0000
cd*     at atapibus? drive ? flags 0x0000
```

Figure 3.26: **SCSI device configuration.** This configuration supports both SCSI and APAPI disks and optical media drives.

can be accomplished with a hardware IOMMU unit, but at least currently they are not available on all standard computer systems.

Due to USBDI, USB drivers do not perform DMA operations. Instead, all DMA operations are done by the host controller on behalf of the devices. We only need to be able to handle DMA requests in the host controller. Since the ugenhc host controller passes requests to the host kernel's ugen driver, there is no need to support DMA in a rump kernel for the purpose of running unmodified USB drivers.

Drivers will still allocate DMA-safe memory to pass to the host controller so that the host controller can perform the DMA operations. We must be able to correctly emulate the allocation of this memory. In NetBSD, all modern device drivers use the machine independent *bus dma* [107] framework for DMA memory. bus dma specifies a set of interfaces and structures which different architectures and busses must implement for machine independent drivers to be able to use DMA memory. We implement the bus dma interface in `sys/rump/librump/rumpdev/rumpdma.c`.

3.10.4 USB Hubs

A ugen can access only the specific USB device it is attached to; this prevents for example security issues by limiting access to one device on the USB bus. For USB functions, such as mass memory or audio, there are no implications. However, USB hubs expose other USB devices (functions or other USB hubs) further down the bus. If a USB hub is attached as ugen, it is possible to detect that devices are attached to the hub, but it is not possible to access any devices after the USB hub, including ones directly attached to it – all ugen access will be directed at the hub. Figure 3.27 illustrates the matter in terms of a device tree, while Figure 3.28 and Figure 3.29 present the same situation in terms of kernel bootlogs. In practice dealing with hubs is not an issue: the host needs to prioritized HUBs over ugen.

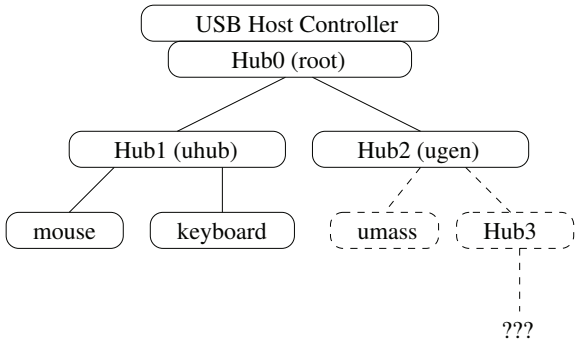

Figure 3.27: **Attaching USB Hubs.** In case a USB hub is attached by the host kernel as *ugen* instead of as a hub, devices tree behind the hub will not be visible.

host probe:

```
ugen2 at uhub1 port 1
ugen2: OnSpec Generic USB Hub
```

rump kernel probe:

```
ugenhc2 at mainbus0
usb2 at ugenhc2: USB revision 2.0
uhub2 at usb2
uhub2: 1 port with 1 removable
uhub3 at uhub2 port 1: OnSpec Inc.
uhub3: 2 ports with 0 removable
uhub4 at uhub3 port 1: OnSpec Inc.
uhub5 at uhub3 port 2: OnSpec Inc.
```

Figure 3.28: **USB device probe *without* host HUBs.**

host probe:

```
uhub5 at uhub2 port 1: OnSpec Generic Hub
uhub5: 2 ports with 0 removable
ugen2 at uhub5 port 1
ugen2: Alcor Micro FD7in1
ugen3 at uhub5 port 2
ugen3: CITIZEN X1DE-USB
```

rump kernel probe:

```
ugenhc2 at mainbus0
usb2 at ugenhc2: USB revision 2.0
uhub2 at usb2
umass0 at uhub2
umass0: Alcor Micro
umass0: using SCSI over Bulk-Only
scsibus0 at umass0
sd0 at scsibus0
sd0: 93696 KB, 91 cyl, 64 head, 32 sec, 512 bytes/sect x 187392 sectors
sd1 at scsibus0
sd1: drive offline
ugenhc3 at mainbus0
usb3 at ugenhc3: USB revision 2.0
uhub3 at usb3
umass1 at uhub3
umass1: using UFI over CBI with CCI
atapibus0 at umass1
sd2 at atapibus0 drive 0
sd2: 1440 KB, 80 cyl, 2 head, 18 sec, 512 bytes/sect x 2880 sectors
```

Figure 3.29: **USB device probe *with* host HUBs.** The devices behind the hub are visible.

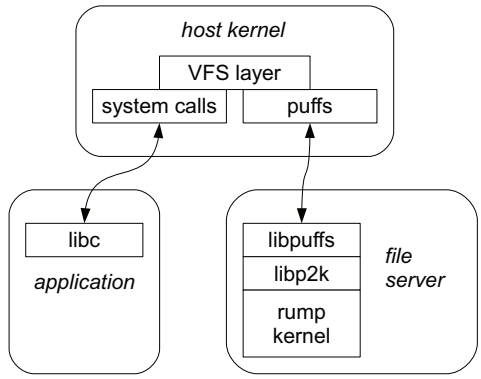

Figure 3.30: **File system server.** The request from the microkernel client is transported by the host kernel to the rump kernel running providing the kernel file system driver. Although only system calls are illustrated, page faults created by the client may be handled by the server as well.

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* [53]. 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 3.30. 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. 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 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 3.31. 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

in-kernel mount:

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

equivalent rump kernel file server mount:

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

Figure 3.31: **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.

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 [71] 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 refer-

```
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 3.32: **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.

ences 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* at A–12). 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 3.32. 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()`.

The `RUMP_VOP_LOCK()` and `RUMP_VOP_UNLOCK()` macros in p2k to deal with NetBSD kernel virtual file system 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 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.

3.12 Remote Clients

Remote clients are clients which are disjoint from the rump kernel. In a POSIX system, they are running in different processes, 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,

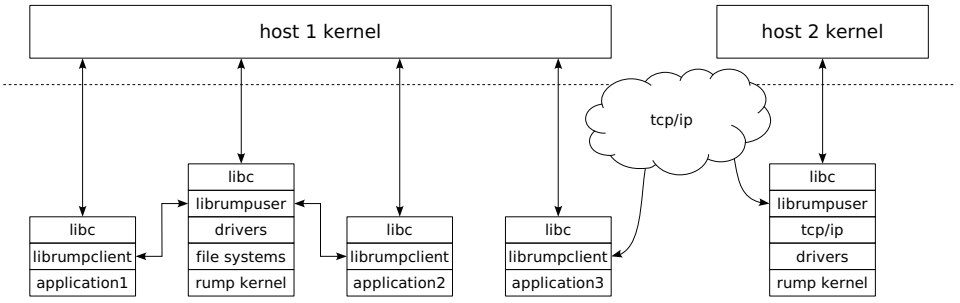

Figure 3.33: **Remote client architecture.** Remote clients communicate with the rump kernel through the *rumpclient* library. The client and rump kernel may or may not reside on the same host.

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 binaries to use services from a rump kernel without recomplication. For example, it is possible to configure an unmodified Firefox browser to use a TCP/IP stack provided by a rump kernel.

The general architecture of remote rump clients is illustrated in Figure 3.33. It is explained in detail in the following sections.

Communication is done using host sockets. Currently, two protocol families can be used for client-server communication: 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.

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 machine.

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 [21] are possible, they are beyond the scope of this dissertation, 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* at A–27). Linking can happen either when a program is compiled or when it is run. 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 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 good example is Firefox, 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 clients can still be used as rump 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(url)` from the local client. The argument the routine takes 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* at A–5) is provided with NetBSD with the standard distribution for serving remote clients. The factions and

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 3.34: **Example invocations lines of `rump_server`.** All invocations create rump kernels listening for clients at different addresses with different capabilities.

drivers supported by the server 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 3.34 illustrates server usage with examples.

The data transport and protocol layer for remote clients is implemented entirely within the hypervisor. Because host sockets were used already, those layers were convenient to implement there. This 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 communication protocol we will discuss in detail next (Section 3.12.4); we will not discuss the interfaces. The interfaces are defined in `sys/rump/include/rump/rumpuser.h` and are implemented in `lib/librumpuser/rumpuser_sp.c` for the hypervisor and `sys/rump/librump/rumpkern/rump.c` for the kernel.

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. By convention, only 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.

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.

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.

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.

Now that we know the communication protocol, we will compare the operations executed in the regular case and in the rump 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.

3.12.5 Of Processes and Inheritance

The process context for a remote client is controlled by the rump kernel server and the rump_lwproc interfaces available for local clients (manual page *rump_lwproc.3* at A–23) 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.

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 "raise" signal model.

Table 3.6: **Requests from the kernel to the client.**

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 marshalls 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.**

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 [79] and TLS [97] with clientside certificates. Implementing any of these mechanisms was beyond the scope of our work.

When a connection is severed, 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 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 syscall. 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

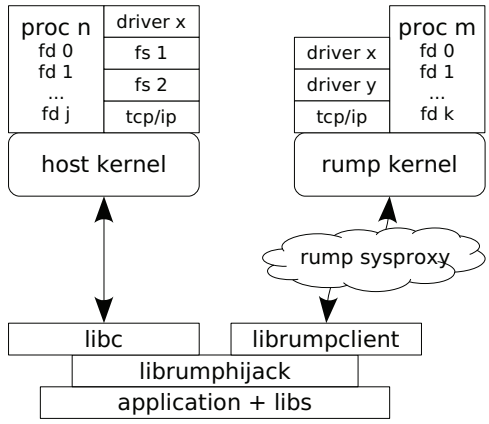

Figure 3.35: **System call hijacking.** The *rumphijack* library intercepts system calls and determines whether the syscall request should be sent to the rump kernel or the host kernel for processing.

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 3.35) 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 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.

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 by rumpclient. Currently they include the communication socket file descriptor and the kqueue descriptor which rumpclient uses for I/O multiplexing. Recall, the socket 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 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.

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 [90] `sshd` or the mutt [83] 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 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 3.36. 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.

```
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 3.36: **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.

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 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.

3.12.8 Performance

Figure 3.37 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.

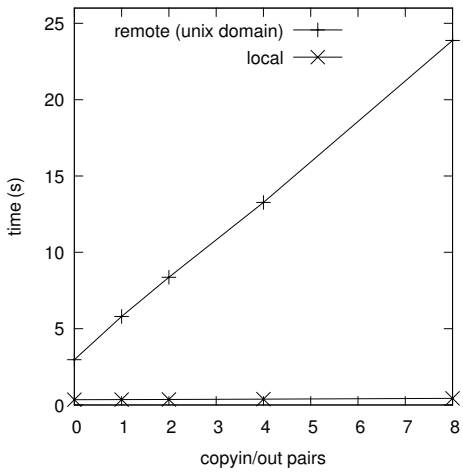

Figure 3.37: **Local vs. Remote system call overhead.** The cost of remote system calls is dominated by the amount of client-kernel roundtrips necessary due to copying data in and out. For local clients the cost of a system call is virtually independent of the amount of copies in and out.

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 of speeds up to 10Mbps, 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 [14, 63] may be attempted.

3.13 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 the following techniques: code moving, function pointers and weak symbols. 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.

4 Evaluation

This chapter evaluated the work in a broad scope. First, evaluate the general feasibility of the implementation. After that, we evaluate rump kernels as the solutions for the motivating problems we gave in Section 1.1: development, security and reuse of kernel code in applications. Finally, we look at performance metrics to measure that rump kernels perform better than other solutions for our use cases.

4.1 Feasibility

The feasibility of an implementation needs to take into account both the initial effort and the maintenance effort. Both must be reasonably low for an implementation to be feasible over a period of time.

$$feasibility = \frac{1}{implementation + maintenance} \tag{4.1}$$

4.1.1 Implementation Effort

To measure implementation effort we compare the size of the supported driver codebase against amount of code required for rump kernel support. The complete reimplemented part along with a selection of supported drivers is presented in Figure 4.1. The extracted drivers depicted represent only a fraction of the total supported drivers. In most cases, smaller drivers were included and larger ones were left out; for example, the omitted FFS driver is three times the size of the FAT driver. From this data we observe that the amount of driver code supported by a rump kernel is far greater than the amount of code implemented for supporting them.

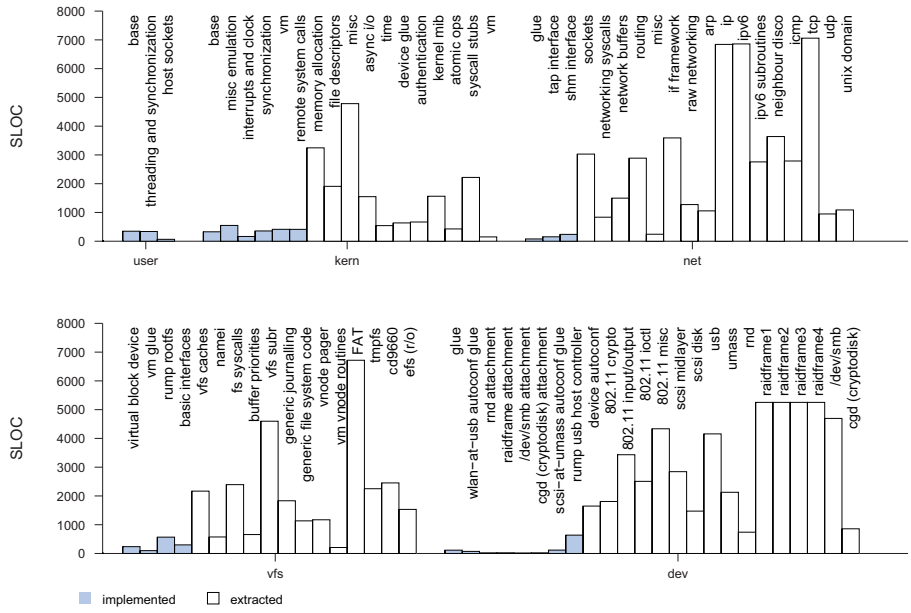

Figure 4.1: **Source Lines Of Code in rump kernel and selected drivers.** The amount of code usable without modification is vastly greater than the amount of support code necessary. A majority of the drivers tested to work in rump kernels is not included in the figure due to the limited space available for presentation. The size difference between rump kernel specific support code and drivers is evident.

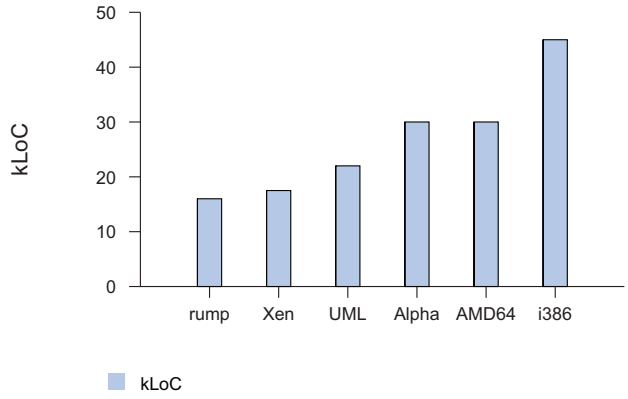

Figure 4.2: **Lines of code for platform support.** All listed platforms with the exception of User Mode Linux (UML) are NetBSD platforms. The figure for UML was measured from the Linux kernel sources.

To get another perspective on the amount of code, the total number of lines for code and headers under `sys/rump` is 16k. This directory not only contains the code implemented for the rump kernel base and factions, but also the drivers implemented for a rump kernel (e.g. the if_virt networking interface which was described in Section 3.9.1). Out of the code in `sys/rump`, 6k lines are autogenerated (system call wrappers alone are 4.5k lines of autogenerated code), leaving 10k lines as implemented code. From the remaining 10k lines, a fifth (2k lines) consists of the rump root file system (Section 3.7) and the USB ugen host controller (Section 3.10).

For comparison, we present the amount of code for full kernel support in Figure 4.2. Given that architecture support is commonly developed with the copy-and-modify approach, looking purely at the figure for lines of code does not give a full estimate of implementation effort. For example, normal architectures include the standard MI VM and interface it to the platform with the pmap module. In contrast, in a rump kernel we provide a partial rewrite for the MI VM interfaces. However, these comparisons give us a rough idea of how much code is required.

Total commits to the kernel	9,640
Total commits to sys/rump	438
Commits touching only sys/rump	347
Build fixes	17
Unique committers	30

Table 4.1: Commit log analysis for `sys/rump` Aug 2007 - Dec 2008.

4.1.2 Maintenance Effort

After support has been implemented, it must be maintained or it will *bitrot* due to code in the rest of the system changing over time. An example of this maintenance in a rump kernel is as follows: an UVM interface is changed, but the implementation in the rump kernel is not changed. If this change is committed, the rump VM implementation still uses the old interface and will fail to build. Here we look at how often the build of NetBSD was broken due to unrelated changes being committed and causing problems with rump kernels.

To evaluate build breakage, we manually examined NetBSD commit logs from August 2007 to December 2008. Support for rump kernels was initially committed in August 2007, so this period represents the first 17 months of maintenance effort, and was the most volatile period since the concept was new to the NetBSD developer community. The findings are presented in Table 4.1.

To put the figure of one problem per month into context, we examine the i386 port build logs [40]: the NetBSD source tree has 15–20 build problems in a typical month. When looking at the total number of commits, we see that 99.8% of kernel commits did not cause problems for rump kernels. We conclude that maintaining requires non-zero effort, but breakage amounts to only a fraction of total breakage.

build.sh rumptest

When rump kernel support was initially added, the NetBSD developer community criticized it because it made checking kernel changes against build problems in unrelated parts of the kernel more time consuming. Before, it was possible to perform the test by building a kernel. After support was added, a more time consuming full build was required.

A full build was required since linking a rump kernel depends on the rumpuser hypercall library which in turn depends on libc. Therefore, it is not possible to build and link a rump kernel without building at least a selection of userspace libraries. More so, it was not obvious to the developer community what the canonical build test was.

Using the NetBSD crossbuild tool *build.sh* [77] it is possible to [cross]build a kernel with a single command. A popular kernel configuration for build testing is the i386 *ALL* configuration, which can be built by invoking the build.sh tool in the following manner:

`build.sh -m i386 kernel=ALL`.

We wanted something which made testing as fast and easy as doing a buildtest with build.sh. We added a `rumptest` subcommand to build.sh. The command crossbuilds rump kernel components. It then passes various valid rump component combinations directly to `ld`. Since these sets are missing the rumpuser library, linking will naturally be unsuccessful. Instead, we monitor the error output of the linker for unresolved symbols. We use the closure of the rump kernel C namespace to locate any in-kernel link errors. If there are some, we display them and flag an error. Otherwise, we signal a successful test.

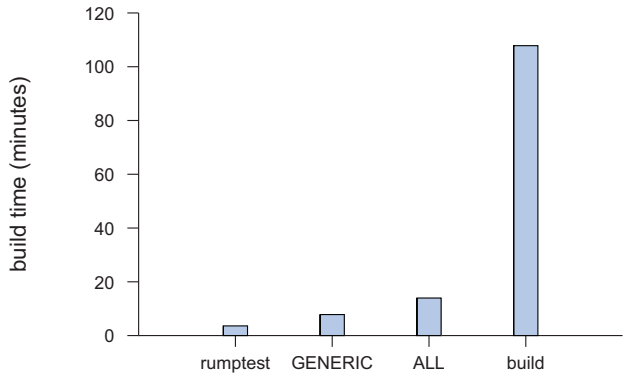

Figure 4.3: **Duration for various i386 target builds.** The targets with names written in capital letter are kernel-only builds for popular test configurations. Prior to the introduction of the *rumptest* target, the *build* target was the fastest way to check that modifications to kernel code did not introduce build problems for rump kernels.

The rumptest command is not a substitute for a full OS build, but it addresses a practical concern of attaining "99%" certainty of the correctness of kernel changes in 1/30th of the time required for a full build. Build times are presented in Figure 4.3.

The addition of the rumptest command addressed the problem: it made build-testing quick and obvious.

4.2 Use of Rump Kernels in Applications

4.2.1 fs-utils

Userspace implementations of file system drivers exist. For example, there is mtools [89] for FAT file systems, e2fsprogs [2] for ext2/3/4, isoread for CD9660 and ntfsprogs

for NTFS. The benefit is that file systems can be accessed in userspace without kernel support or privileges. The downside is that all tools use different command line arguments. Another drawback is that if a userspace driver for a file system is not implemented, such a utility does not exist.

The fs-utils project [116] attached standard NetBSD file utilities (`ls`, `cat`, `mv`, etc.) to rump kernel file system drivers as local clients. There are two benefits to this approach:

1. standard command line parameters are preserved i.e. `ls` accepts the familiar `-ABcFhL` parameter string

2. all NetBSD file systems with kernel drivers are supported

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 [14].

We conclude it is possible to use rump kernels as application libraries to implement functionality which was previously done using userspace reimplementations of drivers.

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

4.2.2 makefs

For a source tree to be fully cross-buildable with build.sh [77], the build process cannot rely on any non-standard kernel functionality because the functionality might not exist on a non-NetBSD build host. The build.sh mandate also demands that a build can be performed fully unprivileged, i.e. a root account is not required.

Before build.sh, the canonical approach to building a file system image for boot media was to be to create a regular file, mount it using the loopback driver, copy the files to the file system and unmount the image. This approach was not compatible with the goals of build.sh.

When build.sh was introduced to NetBSD, it came with a tool called *makefs* which creates a file system image from a given directory tree using only application level code. In other words, the makefs application contains the file system driver implemented for a userspace environment. This approach does not require privileges to mount a file system or support of the target file system in the kernel. The original utility had support for Berkeley FFS and was implemented by modifying and reimplementing the FFS kernel code to be able to run in userspace. This copy-and-modify approach was the only good approach available at the time.

The process of makefs consists of four phases:

1. scan the source directory
2. calculate target image size based on scan data
3. create the target image
4. copy source directory files to the target image

	original	rump kernel backend
FFS SLOC	1748	247
supported file systems	FFS	FFS, ext2, FAT, SysVBFS
FFS effort	> 2.5 weeks (100 hours)	2 hours
total effort	7 weeks (280 hours)	2 days (16 hours)

Table 4.2: `makefs` implementation effort comparison.

In the original version of makefs all of the phases were implemented in a single C program. Notably, phase 4 is the only one that requires a duplicate implementation of features offered by the kernel file system driver.

We implemented makefs using a rump kernel backend for phase 4. We partially reused code from the original makefs, since we had to analyze the source tree to determine the image size (phases 1&2). We rely on an external newfs/mkfs program for creating an empty file system image (phase 3). For phase 4 we use fs-utils to copy a directory hierarchy to a file system image.

For phase 3, we had to make sure that the mkfs/newfs utility can create an empty file system in a regular file. Historically, such utilities operate on device special files. Out of the supported file systems, we had to add support for regular files to the NetBSD FAT and SysVBFS file system creation utilities. Support for each was approximately 100 lines of modification.

We obtained the implementation effort for the original implementation from the author [76] and compare the two implementations in Table 4.2. As can be observed, over a third of the original effort was for implementing support for a single file system driver. Furthermore, the time cited is only for the original implementation and does

not include later debugging [76]. Since we reuse the kernel driver, we get the driver functionality for free. All of the FFS code for the rump kernel implementation is involved in calculating the image size and was available from makefs. If code for this calculation had not been available, we most likely would have implemented it using shell utilities. However, since determining the size involves multiple calculations such as dealing with hard links and rounding up directory entry sizes, we concluded that reusing working code was a better option.

The makefs implementation with a rump kernel backend is available from the *othersrc* module at `othersrc/usr.sbin/makefs-rump`. It also uses the utility at `othersrc/usr.sbin/makefs-analyzetree`. Note that the `othersrc` module is separate from the `src` module we mentioned in Section 1.6.1, and the code must be retrieved from the NetBSD source repository separately.

Interested readers are also invited to look at Appendix B.4, where we present another method for makefs functionality using standard system utilities as remote clients for rump kernel servers.

4.3 On Portability

There are two things to consider with portability. First, given that the NetBSD kernel was ported to run on a hypercall interface, what are the practical implications for hosting a NetBSD rump kernel on either a non-NetBSD system or a NetBSD system of a different version. We call these systems *non-native*. Second, we want to know if the concept of rump kernel construction is unique to the NetBSD kernel codebase, or if it can be applied to other operating systems as well.

4.3.1 Non-native Hosting

When hosting NetBSD kernel code in a foreign environment, there are a number of practical issues to consider:

1. Building the source code for the target system.
2. Accessing the host services from a rump kernel.
3. Interfacing with the rump kernel from clients.

For each subpoint, we offer further discussion based on experiences. We have had success with non-native hosting of NetBSD rump kernels on Linux, FreeBSD and various versions of NetBSD, including the NetBSD 4 and NetBSD 5 branches.

Building

First, we define the build host and the build target. The build host is the system the build is happening *on*. The build target is the system that the build is happening *for*, i.e. which system will be able to run the resulting binaries. For a kernel, the target involves having a compiler targeted at the correct CPU architecture. For a userspace program, it additionally requires having access to the target headers and libraries for compiling and linking, respectively.

The build tool *build.sh* we already mentioned in Section 4.1.2 and Section 4.2.2 automatically builds the right tools and compiler so that the NetBSD source tree can be built for a NetBSD target architecture on any given host. What we want is slightly different: a compilation targeted for a non-NetBSD system. Having build.sh provide

the necessary tools such as *make*[15] and *mtree* is helpful. These tools can be used, but the compiler should be replaced with the foreign host targeted compiler. Also, when building for a different version of NetBSD build.sh is helpful since occasionally features are implemented in the build tools after which the build process starts depending on them. Older versions of NetBSD may not be able to build newer versions using native tools alone.

The *pkgsrc* packaging system [5] provides support for building a rump kernel and the rumpuser hypervisor library on Linux and FreeBSD. Support is provided by the package in `pkgsrc/misc/rump`. While as of writing this text the rump kernel version support from pkgsrc is older than the one described in this document, the package can be used as a general guideline on how to build a NetBSD rump kernel targeted at a foreign host.

Host Services

To be able to run, a rump kernel uses the hypercall interface to access the necessary host services, e.g. memory allocation. We already touched the problems when discussing the hypercall interface in Section 3.2.3. To recap, the rump kernel and the host must agree on the types being passed between each other. Commonly used but not universally constant types such as `time_t` cannot be used as part of the hypercall interface and unambiguous types such as `int64_t` should be used instead. Compound types such as `struct stat` and `struct timeval` cannot generally be used, as the binary layout varies from one system to another — in these cases, both structures contain `time_t` which is enough to make them non-eligible. Furthermore, system-defined macros such as `O_RDONLY` may not be used, since their representation will depend on the system they are from. Since the hypercall interface is relatively small and used only for the specific purpose of interacting with the rump kernel

[15] The "flavour" of `make` required to build NetBSD is different from for example GNU make.

hypervisor, it is possible to construct the interface without relying on forbidden types.

In case any host services which are not standard across operating systems are to be used, a suitable driver must exist within the rump kernel. Examples include raw network access and generic USB access, which may require host-specific handling. Strictly speaking, interfacing with specific host features is a driver issue instead of a rump kernel architectural issue, but it should be kept in mind nonetheless.

Client Interfaces

Client interfaces share the same type problem as the hypercall interface. However, while the hypercall interface is relatively compact, the interfaces available to clients are large — consider the system call interface. Furthermore, while we were in total control of the hypercall interface, we cannot change the system call interface since it is specified by POSIX and various other standards. Finally, many covert protocols exist in the form of structures passed between the client and kernel drivers with calls such as `ioctl()` and `write()`. Undefined operation will result where the type systems between the rump kernel and client do not agree.

Since everything in C is in a flat namespace, it is not possible to include NetBSD headers on for example Linux — there cannot be two different simultaneous definitions for `struct sockaddr`. This is not an issue unique to rump kernels, but rather to all systems which wish to provide an alternative implementation of a standard system feature. One such example is the BSD sockets interface provided by the lwIP TCP/IP stack [30]. Unlike rump kernels currently, lwIP exports the correct types for the clients (e.g. `struct sockaddr`) in its own headers, so as long as clients do not include host headers which supply conflicting definitions, things will work, since both the client and the service will use the same type definitions. This ap-

proach of supplying alternate clientside definitions for all types is unlikely to scale to rump kernels. Consider `time_t` again: the client may not include any header which transitively includes `<sys/types.h>`.

One option for type compatibility is to manually go over the sources and provide compatibility between foreign clients and the kernel. This approach was taken by OSKit [34] to allow parts of kernel code to be hosted on non-native platforms and run inside foreign kernels. The manual approach is uninviting for us. The anykernel architecture and rump kernel should remain fully functional at all times during a NetBSD development cycle instead of having dedicated releases. It is highly unlikely open source developers will provide working translation routines for every change they make, and any approach requiring multiple edits for the same information can be viewed both as a burden and an invitation for bugs.

NetBSD the provides translation of system call parameters for some operating systems such as Linux and FreeBSD under `sys/compat`. The original idea with compat support is to be able to run application binaries from foreign operating systems under a regular NetBSD installation. As such, the compat code can be used for translation of system calls used by regular applications from supported foreign clients, e.g. the sockets interfaces. However, sys/compat support does extend to various configuration interfaces, such as setting the IP address of a networking interface.

In NetBSD, system headers are represented directly with C syntax. If they were written in a higher level markup and the C representation were autogenerated from that, some automatic translation helpers could be attempted. However, such an undertaking is beyond the scope of this work.

Since a complete solution involves work beyond the scope of this project, we want to know if a partial solution is useful. Currently, some compatibility definitions that have been necessary to run NetBSD rump kernels on foreign hosts are provided in

```
fromvers () {
        echo
        sed -n '1{s/\$//gp;q;}' $1
}

fromvers ../../../sys/fcntl.h
sed -n '/#define     O_[A-Z]*        *0x/s/O_/RUMP_O_/gp' \
    < ../../../sys/fcntl.h
```

Figure 4.4: Simple compatibility type generation.

```
/*      NetBSD: fcntl.h,v 1.36 2010/09/21 19:26:18 chs Exp      */
#define RUMP_O_RDONLY   0x00000000      /* open for reading only */
#define RUMP_O_WRONLY   0x00000001      /* open for writing only */
#define RUMP_O_RDWR     0x00000002      /* open for reading and writing */
```

Figure 4.5: Generated compatibility types.

`sys/rump/include/rump/rumpdefs.h`. These definitions are extracted from system headers with regular expressions by a script called `makerumpdefs.sh` (available from the same directory). An example portion of the script is presented in Figure 4.4 and the corresponding result is available in Figure 4.5. When examined in detail, the weakness is that the compatibility type namespace includes neither the OS nor the OS revision. We felt that including that information would make the names too cluttered for the purpose of a quick fix.

Despite there being no complete client interface compatibility, the fs-utils suite runs on Linux [116]. Old NetBSD hosts can use system calls without problems due to system call compatibility. In fact, most of the development work described in this dissertation was done for NetBSD-current on a NetBSD 5 host. While binary compatibility for the vnode interface is not maintained by NetBSD, we were able to continue running the microkernel style file servers after an ABI change by adding compatibility translation to libp2k. We therefore conclude that while foreign clients

are not out-of-the-box compatible with NetBSD rump kernels, specific solutions can be implemented with relative ease and that foreign hosting is possible.

4.3.2 Other Codebases

To investigate adding rump support to other kernels, prototypes of rump kernel support for Linux 2.6 and the FreeBSD 8.0 kernel were implemented. Both prototypes were limited in completeness: for example, the application interfaces were manually constructed instead of being autogenerated like on NetBSD. Still, both could provide services from their respective kernel codebases on a NetBSD host.

The Linux implementation was aimed at running the Journaling Flash File System 2 (jffs2) [113] as a microkernel style file server. This file system made an interesting use case since NetBSD did not have a native file system suitable for flash media back in 2008. The result was a functional jffs2 file server for NetBSD which could read and write files in a file system image file. The amount of effort required for the prototype was two weeks of working time.

The FreeBSD prototype is capable of using the FreeBSD UFS implementation to access the directory namespace in a standalone application. The FreeBSD prototype took two working days to implement. Based on additional studying of the source code and prior experience, it is estimated that 1–2 days of work would allow the prototype to provide full read-only support.

As the hypervisor library, both implementations could use the existing rumpuser hypercall library and no separate implementation was required.

Both prototype implementation experiments gave reason to believe that it is possible to adjust the respective codebases to comply with the anykernel architecture.

4.4 Security: A Case Study with File System Drivers

What happens in userspace stays in userspace!

– Tod McQuillin

As we mentioned in Chapter 1, in a general purpose OS drivers for disk-based file systems are written assuming that file system images contain trusted input. While this assumption was true long ago, in the age of USB sticks and DVDs it no longer holds. Still, users mount untrusted file systems using kernel code. Arbitrary memory access is known to be possible via the use of a suitable crafted file system image and fixing each file system driver to be bullet-proof is at best extremely hard [115].

When run in a rump kernel, a file system driver dealing with an untrusted image is isolated in its own domain thus mitigating an attack. The rump kernel can then be attached to the host file system namespace by mounting it, as described in Section 3.11. This separation mitigates the possibility of a direct memory access attack on the kernel, but is transparent to users.

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 4.6 illustrates what happens when the file 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 should 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

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

Figure 4.6: **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.

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 [52] or sandboxing [35]. Networked file system clients (such as NFS and CIFS) may also benefit from the application of firewalls.

In conclusion, a rump kernel provides security benefits for the tasks it is meant for. But like everything else, it depends on the security of the layers it is running on and cannot make up for flaws in the underlying layers such the host OS, hardware or laws of physics.

4.5 Testing and Developing Kernel Code

Kernel driver development is a form of software development with its own quirks [60]. The crux is the fact that the kernel is the "life support system" for the platform,

be the platform hardware or a hypervisor. If something goes wrong during the test, the platform and therefore any software running on top of it may not function properly. This section evaluates the use of rump kernels for kernel driver testing and debugging in the context of NetBSD drivers.

4.5.1 The Approaches to Kernel Development

We start by looking at the approaches which can be used for driver development and testing. The relevant metric we are interested in is the convenience of the approach, i.e. how much time a developer spends catering to the needs of the approach instead of working on driver development itself. One example of a convenience factor is iteration speed: is iteration nearly instantaneous or does it take several seconds.

Driver-as-an-application

In this approach driver under development is isolated to a self-contained userspace program. As mentioned in Chapter 1, doing the first attempt on the application level is a common way to start driver development. The driver is developed against a pseudo-kernel interface.

There are three problems with this approach. First, the emulation is often lazy and does not reflect kernel reality and causes problems when the driver is moved to run in the kernel. Consider development in the absence of proper lock support: even trivial locking may be wrong, not to mention race conditions. Second, keeping the userspace version alive after the initial development period may also prove challenging; without constant use the `#ifdef` portion of the code has a tendency to bitrot and go out-of-sync because changes to the kernel. Finally, if this technique is used in multiple drivers, effort duplication will result.

Full OS

By a full OS we mean a system running on hardware or in a virtualized environment. In this case the driver runs unmodified. For some tasks, such as development of hardware specific code without access to a proper emulator, running on raw iron is the only option. In this case two computers are typically used, with one used for coding and control, and the other used for testing the OS. Booting the OS from the network or other external media is a common approach, since that source media avoids an extra reboot to upload an adjusted kernel after a fix. Some remote debugging aids such as gdb-over-Ethernet or FireWire Remote Memory access may be applicable and helpful.

The other option is to run the full OS hosted in a virtualized environment. Either a paravirtualization such a Xen [11] or a hardware virtualization such as with QEMU [13] may be used. Virtualized operating systems are an improvement over developing directly on hardware since they avoid the need of dealing with physical features such as cables.

Using a full OS for development is not as convenient as using an isolated userspace program. We justify this statement by reduction to the absurd: if using a full OS were as convenient for development as using an isolated application, nobody would go through the extra trouble of building an ad-hoc userspace shim for developing code as an isolated userspace application. Since ad-hoc userspace shims are still being built for driver development, we conclude that development as a userspace application is more convenient.

```
==11956== 1,048 (24 direct, 1,024 indirect) bytes in 1 blocks are definitely lost in loss record 282 of 315
==11956==    at 0x4A05E1C: malloc (vg_replace_malloc.c:195)
==11956==    by 0x523044D: rumpuser__malloc (rumpuser.c:156)
==11956==    by 0x5717C31: rumpns_kern_malloc (memalloc.c:72)
==11956==    by 0x570013C: ??? (kern_sysctl.c:2370)
==11956==    by 0x56FF6C2: rumpns_sysctl_createv (kern_sysctl.c:2086)
==11956==    by 0xEFE2193: rumpns_lfs_sysctl_setup (lfs_vfsops.c:256)
==11956==    by 0xEFE26CD: ??? (lfs_vfsops.c:354)
==11956==    by 0x5717204: rump_module_init (rump.c:595)
==11956==    by 0x56DC8C4: rump_pub_module_init (rumpkern_if_wrappers.c:53)
==11956==    by 0x50293B4: ukfs_modload (ukfs.c:999)
==11956==    by 0x5029544: ukfs_modload_dir (ukfs.c:1050)
==11956==    by 0x4C1745E: fsu_mount (fsu_mount.c:125)
```

Figure 4.7: **Valgrind reporting a kernel memory leak.** The memory leak detector of Valgrind found a memory leak problem in kernel code.

Rump Kernels

Rump kernels are fast, configurable, extensible, and can simulate interaction between various kernel subsystems. While almost the full OS is provided, the benefits of having a regular application are retained. These benefits include:

- Userspace tools: dynamic analysis tools such as Valgrind [88] can be used to instrument the code. Although there is no full support for Valgrind on NetBSD, a Linux host can be used to run the rump kernel. Figure 4.7 [50] shows an example of a NetBSD kernel bug found with Valgrind on Linux.

  Also, a debugger such as `gdb` can be used like on other userlevel applications.

- Complete isolation: Changing interface behavior for e.g. fault and crash injection [46, 95] purposes can be done without worrying about bringing the whole system down. The host the rump kernel is running on acts as "life support" just like in the application approach, as opposed to the kernel under test itself like in the full OS approach.

- Rapid prototyping: One of the reasons for implementing the 4.4BSD log-structured file system cleaner in userspace was the ability to easily try different cleaning algorithms [100]. Using rump kernels these trials can easily be done without having to split the runtime environment and pay the overhead for easy development during production use.

Since it is difficult to measure the convenience of kernel development by any formal metric, we would like to draw the following analogy: kernel development on real hardware is to using emulators as using emulators is to developing with rump kernels.

4.5.2 Test Suites

A test suite consists of test cases. A test case produces a result to the question "is this feature working properly?", while the test suite produces a result to the question "are all the features working properly?". Ideally, the test suite involves for example checking no bug ever encountered has resurfaced [57].

The NetBSD testing utility is called the Automated Testing Framework (*ATF*) [75]. The qualifier "automated" refers to the fact that the test suite end user can run the test suite with a single command and expect to get a test report of all the test cases. The framework takes care of all the rest. ATF consists of tools for running tests and generating reports. One goal of ATF is to make tests *easy* to run, so that they are actually run by developers and users.

Automated batchmode testing exposes several problems in the straightforward approach of using the test suite host kernel as the test target:

1. A failed test can cause a kernel panic and cause the entire test suite run to fail.

2. Activity on the host, such as network traffic or the changing of global kernel parameters can affect the test results (or vice versa: the test can affect applications running on the host).

3. The host kernel test must contain the code under test. While in some cases it may be possible to load and unload drivers dynamically, in other cases booting a special kernel is required. Rebooting disrupts normal operation.

4. Automatically testing e.g. some aspects of the NetBSD networking code is impossible, since we cannot assume peers to exist and even less for them to be configured suitably for the test (network topology, link characteristics, etc.).

It is possible to add a layer of indirection and boot a separate OS in a virtual machine and use that as the test target. This indirection, however, introduces several new challenges which are specific to test suites:

1. Test data and results must be transmitted back and forth between the target and the host.

2. Bootstrapping a full OS per test carries an overhead which would severely limit testing capacity. Therefore, test OS should must be cached and managed. This management involves issues like tracking the incompatible features provided by each instance and rebooting an instance if it crashes.

3. Not only kernel crashes, but also the crashes of the test applications running inside the test OS instances must be detected and analyzed automatically.

Solving these challenges is possible, but adds complexity. Instead of adding complexity, we argue that the simplicity of rump kernels is the best solution for the majority of kernel tests.

4.5.3 Testing: Case Studies

We will go over examples of test cases which use rump kernels. All of the tests we describe are run daily as part of the NetBSD test suite. Most of the cases we describe were written to trigger a bug which was reported by a third party. In these cases we include the NetBSD *problem report* (PR) identifier. A PR identifier is of the format "PR category/number". The audit trail of a PR is available from the web at `http://gnats.NetBSD.org/number`, e.g. PR kern/8888 is available at `http://gnats.NetBSD.org/8888`.

We group the test cases we look at according to qualities provided by rump kernels.

Isolation

Isolation refers to the fact that a rump kernel is isolated from the test host. Any changes in the rump kernel will not directly affect the host.

- PR `kern/44196` describes a scenario where the *BPF* [67] driver leaks *mbufs*. A test case in `tests/net/bpf/t_bpf.c` recreates the scenario in a rump kernel. It then queries the networking resource allocation statistics from the rump kernel (equivalent of `netstat -m`). If the number of mbufs allocated is non-zero, we know the driver contains the bug. We can be sure because we know there is absolutely no networking activity we are not aware of in a rump kernel which serves only local clients and does not have world-visible networking interfaces.
- As mentioned earlier in Section 3.8.1, NetBSD kernel modules are either compiled into the kernel memory image or loaded later. Builtin modules may be enabled and disabled at runtime, but cannot be loaded or unloaded.

Standard rules still apply, and the module must have a reference count of zero before it can be disabled — otherwise threads using the driver would find themselves in an unexpected situation. Since the builtin module code path is different from the loadable module path, it must be tested separately. On a regular system there is no builtin module reserved for testing. This lack means that testing requires the booting of a special kernel.

The test in `tests/modules/t_builtin.c` does tests on a rump kernel with a builtin *kernfs* module. Since the rump kernel instance is private to the test, we have full certainty of both the module state and that there are no clients which wish to use it.

- PR `bin/40455` [16] reported a problem when changing a *reject* route to a *blackhole* route with the `route` command. Since the rump kernel's routing table is private and the rump kernel instance used for the test is not connected to any network, the exact same IP addresses as in the PR could be used in the test case in `tests/net/route/t_change.sh`. Not only does using a rump kernel guarantee that running the test will never interfere with the test host's networking services, it also simplifies writing tests, since reported parameters can be directly used without having to convert them to test parameters. Furthermore, artificial testa parameters may eventually turn out to be incorrect when the test is run on a different host which is attached to a different network.

- PR `kern/43785` describes a *scsipi*[17] problem where ejecting a CD or DVD produces an unnecessary diagnostic kernel message about the media not being present. Requiring a CD to be attached to the test host would make automated testing more difficult. We wrote a rump kernel virtual SCSI target suitable for testing purposes. The target driver is 255 lines long (including comments) and is available from `sys/rump/dev/lib/libscsitest`. Using

[16] In the end it turned out that it was "kern" problem instead of "bin".
[17] *scsipi* is a NetBSD term used to describe the unified SCSI/ATAPI mid-layer.

```
static void
scsitest_request(struct scsipi_channel *chan,
        scsipi_adapter_req_t req, void *arg)
{
        [....]

        case SCSI_SYNCHRONIZE_CACHE_10:
                if (isofd == -1) {
                        if ((xs->xs_control & XS_CTL_SILENT) == 0)
                                atomic_inc_uint(&rump_scsitest_err
                                    [RUMP_SCSITEST_NOISYSYNC]);
                        sense_notready(xs);
                }
                break;

        [....]
}
```

Figure 4.8: **Flagging an error in the scsitest driver.** The test is run with the test program as a local client and the error is flagged. After the test case has been run, the test program examines the variable to see if the problem triggered. Direct access to the rump kernel's memory avoids having to return test information out of the kernel with more complex interfaces such as *sysctl*.

the test driver it is possible to serve a host file as media and replicate the bug. Detecting the scsipi bug is illustrated in Figure 4.8.

The benefits of using a rump kernel over a loadable module on a regular kernel are as follows. First, detecting the error in the test case is a simple matter of pointer dereference instead of having to use a proper interface such as *sysctl* for fetching the value. This ability to do kernel memory access in the test makes writing both the test driver and test case simpler. Second, the virtual CD target used for testing is always `cd0` regardless of the presence of CD/DVD devices on the host. Once again, these benefits do not mean testing using other approaches would be impossible, only that testing is more convenient with a rump kernel.

Multiplicity

Multiplicity means it is possible to run an arbitrary number of kernels and is an important feature especially for networking tests. It allows running tests on for example routing code by simulating a network with n arbitrarily connected hosts.

- PR `kern/43548` describes a kernel panic which happens under certain configurations when the `ip_forward()` routing routine sends an ICMP error for a suitably crafted packet. The test case in `tests/net/icmp/t_forward.c` forks two rump kernels: one is a router and the other one sends the triggering packet. Both rump kernels are configured accordingly. When the triggering party has finished configuration, it immediately sends the triggering packet. Since the test setup uses *shmif*, we do not have to wait for the router to be configured — the router will receive the packet from the shmif bus as soon as it is configured.

  The test monitors the status of the process containing the router. If the bug is triggered, the result is a kernel panic. This panic causes the process to dump core, the test case to notice it, and the test to fail.

- The Common Address Redundancy Protocol (CARP) [86] protocol allows several hosts to handle traffic to the same IP address. A common usage scenario is a hot spare for a router: if a router goes down, a preconfigured hot spare will take over. The CARP protocol allows the hot spare unit to detect the failure via timeout and assume control of the IP address of the service.

  A test in `tests/net/carp` tests the in-kernel hot spare functionality with a straightforward approach. First, two rump kernels are initialized and a common IP address is configured for them using CARP. After verifying that the master unit responds to ping, it is killed by sending `SIGKILL`. The test

waits for the timeout and checks that the address again responds to ping due to the spare unit having taken over the IP address.

Safety

Safety means that a test cannot crash the host. The property is closely related to isolation, but is not the same thing. For example, container/jails based virtualization can provide isolation but not safety. The safety property of a rump kernel is useful especially for inserting panic-inducing test cases into the test suite immediately after they are reported and for test-driven driver development.

- PR `kern/36681` describes a locking inversion in the tmpfs rename routine. The author is not aware of anyone hitting the problem in real life, although, since the effect was a hang instead of an explicit kernel panic, the problem may have gone unreported. Nevertheless, the problem triggered easily with a test program when run on a rump kernel with more than one virtual CPU configured, due to parts of the execution having to interleave in a certain way (the number of host CPUs did not have high correlation).

  A test case in `tests/fs/tmpfs/t_renamerace.c` creates two worker threads which try to trigger the lock inversion. After n seconds the test case signals the worker threads to exit and tries to join the threads. If the threads join successfully, the test passes. In case the test has not passed within $n + 2$ seconds, it timeouts and is declared failed on the grounds that the worker threads have deadlocked and are therefore unable to exit.

- Every file system driver is not of the same maturity and stability. For example, the FFS and FAT drivers are stable, while LFS is experimental. Since all file systems can be tested for functionality via the same system calls, it goes to reason it should be done as much as possible to minimize the amount

of tests that have to be written. The file system independent testing facility located in `tests/fs/vfs` [49] applies the same test to a number of file systems regardless of perceived stability. If an experimental driver causes a rump kernel crash, the failure is reported and the test run continues.

The file system independent rump kernel test suite can also serve as a set of test cases for file system driver development. The development phase is when kernel panics are most frequent. The time taken to execute the tests is not affected by the number of test cases ending in a kernel panic. Since testing uses the rump kernel system call interface and the file system under test is being accessed exactly like in a regular kernel, the tests can be expected to provide a realistic report of the file system driver's stability.

Easily obtainable results

- The `swwdog` software watchdog timer is an in-kernel watchdog which reboots the kernel unless the watchdog is *tickled* often enough by an application program. We wanted to test if the swwdog driver works correctly by checking that it reboots the system in the correct conditions.

  A test case in `tests/dev/sysmon/t_swwdog.c` forks a rump kernel. If the rump kernel exits within the timeout period, the test passes. Otherwise, the test is flagged a failure. The result of the reboot is obtained simply by calling `wait()` for the child and examining the exit status. To obtain the test result, we also need to check that the kernel under test did not reboot when the watchdog was being tickled.

  Writing this test caused us to discover that the watchdog driver could not perform a reboot. It attempted to call reboot from a soft interrupt context, but that had been made illegal by kernel infrastructure changes after the watchdog was initially implemented.

- We noticed that even in superficially identical setups, such as installations of the same OS version in QEMU, different results could be produced. An example case of such a scenario was a test involving `rename()` on a FAT file system. In some setups the test would simply panic the kernel with the error "stack size exceeded" [40]. It was very difficult to start to guess where the error was based purely on this information.

  To make it easier to find problems, we adjusted the ATF test tool to automatically include a stack trace in the report in case the test case dumped a core. Since a rump kernel runs as a regular application and a kernel panic causes a regular core dump, no special support is required for extracting the kernel stack trace. The contents of the test report which motivated this change are presented in Figure 4.9. The tests are run against a production build of the binaries with compiler optimizations turned on, and the stacktrace needs to be examined with that in mind. The information from the stack trace allowed us to fix an off-by-one buffer size problem in the FAT driver's rename routine [18].

Low overhead

Here we do not look at any particular test case, but rather the NetBSD test suite as a whole. The test suite in 5.99.48 contains a total of 2,053 cases. These test cases bootstrap 911 rump kernels, with each test case using zero, one, or up to 16 rump kernels. Running the entire test suite in a NetBSD instance hosted in *qemu-kvm* took 56 minutes on March 31st, 2011 [40]. For comparison, if we assume that bootstrapping one virtualized OS for testing were to take 4 seconds, bootstrapping 911 instances would take over an hour; this overhead is more than what it took to run the entire test suite.

[18] revision 1.72 of `sys/fs/msdosfs/msdosfs_vnops.c`

```
test program crashed, autolisting stacktrace:
(no debugging symbols found)
Core was generated by `t_vnops'.
Program terminated with signal 6, Aborted.
#0  0xbb8eee57 in _lwp_kill () from /usr/lib/libc.so.12
#0  0xbb8eee57 in _lwp_kill () from /usr/lib/libc.so.12
#1  0xbb8eee15 in raise () from /usr/lib/libc.so.12
#2  0xbb8d3bfa in __nrv_alloc_D2A () from /usr/lib/libc.so.12
#3  0xbb8d3c4e in __stack_chk_fail () from /usr/lib/libc.so.12
#4  0xbb8d3c68 in __stack_chk_fail_local () from /usr/lib/libc.so.12
#5  0xbbb6e555 in rumpns_msdosfs_rename () from /usr/lib/librumpfs_msdos.so.0
#6  0xbb997e90 in rumpns_VOP_RENAME () from /usr/lib/librump.so.0
#7  0xbba225d7 in rumpns_do_sys_rename () from /usr/lib/librumpvfs.so.0
#8  0xbba226c6 in rumpns_sys_rename () from /usr/lib/librumpvfs.so.0
#9  0xbb9c08ff in rumpns_sys_unmount () from /usr/lib/librump.so.0
#10 0xbb9c3c14 in rump___sysimpl_rename () from /usr/lib/librump.so.0
#11 0x08056b87 in rename_dir ()
#12 0x08063059 in atfu_msdosfs_rename_dir_body ()
#13 0x0807b34b in atf_tc_run ()
#14 0x0807a19c in atf_tp_main ()
#15 0x0804c7a2 in main ()
stacktrace complete
```

Figure 4.9: **Automated stack trace listing.** The testing framework *ATF* was modified to produce a stack trace for crashed test programs. In case a test against a rump kernel ends in a crash or kernel panic, the kernel stack trace is automatically included in the test report.

As a further demonstration of how lightweight rump kernels are, we point out that the test suite run was completed in a QEMU instance which was limited to 32MB of memory [40].

4.5.4 Regressions Caught

Next we present examples of regressions in NetBSD that were caught by the test suite. We limit these observations to test cases which use rump kernels, and bugs which were caught by a pre-existing test case after a commit was made [40]. Bugs

which were discovered during writing tests or which were found by running the test suite prior to commit are not included. Arguably, the full test suite should always be run prior to commit so that regressions would never hit the public tree, but doing so is not always practical for small changes.

- VFS changes causing invariants to no longer be valid.
- Leak of vnode objects by changes to file system rename routines.
- Regression of a BPF fix which would cause it to accept programs which execute a divide-by-zero.
- Locking problem where the kernel giant lock was released in the networking stack although it was not held.

Although the test suite in 5.99.48 covers only a fraction of the kernel, it is capable of detecting real regressions. As tests accumulate, this capability will increase.

Additionally, the tests using rump kernels stress the host system enough to be the only tests to catch some host bugs, such as a kernel race condition from when TLS support was introduced to NetBSD. In case tests are hosted on a NetBSD system, multiple layers of bugs in NetBSD will be exercised: bugs in the host system, bugs in the kernel code under test in a rump kernel and bugs in rump kernel support. The benefits of using rump kernels for testing do not apply to the first set of bugs, which again shows what we already mentioned in Section 4.4: a rump kernel cannot make up for the deficiencies of layers below it.

4.5.5 Development Experiences

Since a rump kernel is slightly different from a regular kernel e.g. with VM, it is valid to question if kernel code developed in a rump kernel can be expected to run in a regular kernel. The author has been using rump kernels for kernel development since mid-2007 and has not run into major problems.

One observed problem with using a rump kernel is that omitting the `copyin()` or `copyout()` operation for moving data between the application and kernel goes unnoticed with local clients. However, the same phenomenon happens for example on i386 architecture on NetBSD due to the kernel being mapped on top of the current process and code needs to be fixed afterwards [19]. Therefore, this issue is not unique to rump kernels.

Differences can also be a benefit. Varying usage patterns expose bugs where they were hidden before. For example, NetBSD problem report `kern/38057` described a FFS bug which occurs when the file system device node is not on FFS itself, e.g. `/dev` is on tmpfs. Commonly, `/dev` is on FFS, so regular use did not trigger the problem. However, when using FFS in a rump kernel the device node is located on rumpfs and the problem triggers more easily. In fact, this problem was discovered by the author while working on the file system journaling support by using rump file systems.

We conclude that we do not expect any more problems between rump kernels and regular kernels than between various machine architectures. Some problems are caught with one test setup and others are caught with another type of test setup.

[19] e.g. rev. 1.9 of `sys/kern/sys_module.c`.

version	swap	no swap
NetBSD 5.1	17MB	20MB
NetBSD 5.99.48	18MB	24MB

Table 4.3: **Minimum memory required to boot the standard installation.** The memory represents the amount required by the guest. The amount of host memory consumed may be higher due to virtualization overhead.

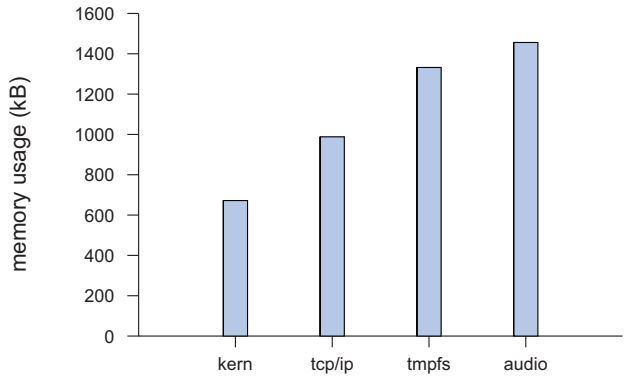

Figure 4.10: **Memory usage of rump kernels per idle instance.** The figures represent the amounts of memory used on the host.

4.6 Performance

Last, we look at performance figures for rump kernels. Examples of the metrics include syscall overhead, memory footprint and bootstrap time. We compare results against other virtualization technologies and against the native system. Even though the anykernel makes it possible to run the same drivers in the monolithic kernel as well as rump kernels, the performance of a rump kernel needs to be on a level where it does not hinder the execution of the intended use cases. Furthermore, we are interested in seeing if rump kernels can outperform other virtualization technologies.

4.6.1 Memory Overhead

We define memory overhead as the amount of memory required by the virtualized OS. We analyze the amount of memory required to successfully boot a standard NetBSD installation up to the root shell prompt in less than 5 minutes. The results are presented in Table 4.3. They were obtained by running `anita install` on the release build and testing if the system boots with different values for `qemu -m`, where the `-m` parameter controls the amount of hardware memory presented to the guest.

A big difference between 5.1 and 5.99.48 is that the latter uses a partially modular kernel by default. Modularity means that all drivers are not loaded at bootstrap, but rather on-demand as they are required. On-demand loading in turn means that the memory requirement of the default installation of 5.99.48 is closer to what is actually required by an application, since fewer unnecessary drivers are loaded into unpageable kernel memory. However, it is not surprising that 5.99.48 requires more memory due to the tendency of software to grow in size.

Still, the default installation may not represent the true minimum required by an application. If we estimate that the memory consumption of NetBSD can be brought down to 1/4th by rigorous source level customization, the true minimum memory required to boot the full OS version of NetBSD 5.99.48 is 4.5MB.

Next, we present the *host* memory consumption for various rump kernel instances in Figure 4.10. Again, we measure the second instance for reasons listed above. In the figure, the *kern* configuration contains nothing but the rump kernel base. The *net* configuration contains TCP/IP drivers. The *tmpfs* configuration supports mounting a tmpfs file system, and the *audio* configuration provides the NetBSD pseudo-audio driver (*pad*). Notably, the audio configuration also includes the vfs faction, since audio devices are accessed via `/dev/audio`.

It needs to be stressed that Table 4.3 only measures the amount of memory used by the virtualized NetBSD instance. The true cost is the amount of memory used on the system hosting the virtualized instance. This cost includes the virtualization container itself. The memory consumption, ignoring disk cache, for the *second* `qemu -n 24` instance for NetBSD 5.99.48 on the host is 38.2MB — measuring the second instance avoids one-off costs, which are not relevant when talking about scaling capability. The difference between what the guest uses and what the host uses in this case is an additional 14.2MB in total memory consumption.

When compared to the memory usage of 38.2MB for a virtual QEMU instance, a rump kernel's default consumption is smaller by a factor of more than 20. This difference exists because a rump kernel does not need to duplicate features which it borrows from the host, and due to its flexible configurability, only the truly necessary set of components is required in the guest.

4.6.2 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 we mentioned in Section 2.1, delays of over 100ms are perceivable to humans [78].

We measured the bootstrap times of a full NetBSD system for two setups, one on hardware and one in a QEMU guest. While bootstrap times can at least to some degree be optimized by customization, the figures in Table 4.4 give us an indication of how long it takes to boot NetBSD. To put the figures into use case context, let us think about the testing setup we mentioned in Section 4.5.3. The test suite boots 911 rump kernels and an entire run takes 56 minutes. Extrapolating from Table 4.4,

platform	version	kernel boot	login prompt
hardware	NetBSD 5.1	8s	22s
QEMU	NetBSD 5.99.48	14s	28s

Table 4.4: Bootstrap times for standard NetBSD installations.

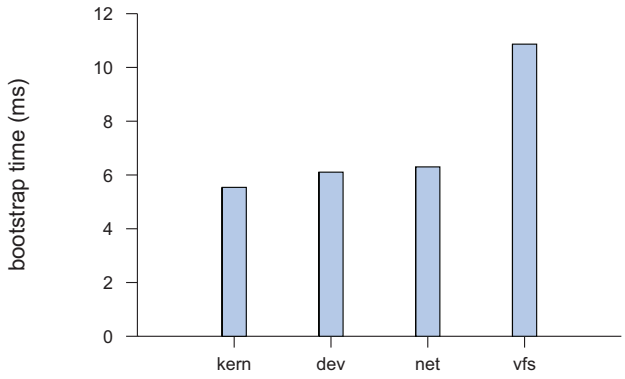

Figure 4.11: Time required to bootstrap one rump kernel. The time varies from configuration to configuration because of the the initialization code that must be run during bootstrap.

bootstrapping 911 instances of NetBSD in QEMU takes 7.1 hours, which is seven and a half times as long as running the entire test suite took using rump kernels.

The bootstrap times for various rump kernel faction configurations are presented in Figure 4.11. 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 system by a factor of 1000 with this metric.

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 [44] by enabling physical hosts to have multiple independent networking stacks and routing tables.

We measure the total time it takes to bootstrap, 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 4.12. 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 used if node startup and configuration is a bottleneck. Both approaches provide the same features during runtime. The results are presented in Figure 4.13.

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

```
#!/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 4.12: **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.

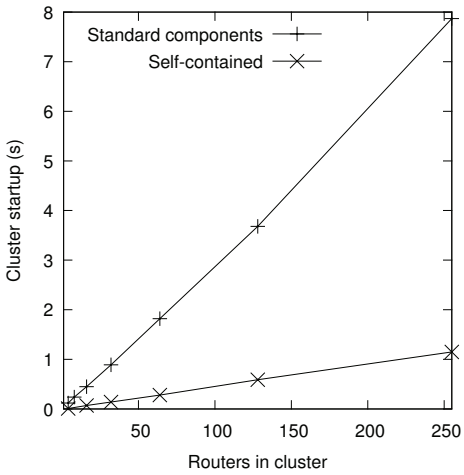

Figure 4.13: **Time required to start, configure and send an initial packet.**

4.6.3 System Call Speed

We compared rump system call performance against other technologies: Xen, QEMU (unaccelerated) and User-Mode Linux. We did this comparison by executing the `setrlimit()` system call 5 million times per thread in two simultaneously running host threads. We ran the UML and Xen tests on a Linux host. For calibration, we provide both the NetBSD and Linux native cases. We were unable to get UML or QEMU to use more than one host CPU. For a NetBSD host we present native system calls, a rump kernel guest, and a QEMU NetBSD guest. For Linux, we have native performance, a Linux Xen guest and a UML guest. The results are presented in Figure 4.14. The NetBSD native call is 16% faster than the Linux native call. We use this ratio to normalize the results when comparing rump kernels against Linux. We did not investigate the reason for the difference between NetBSD and Linux.

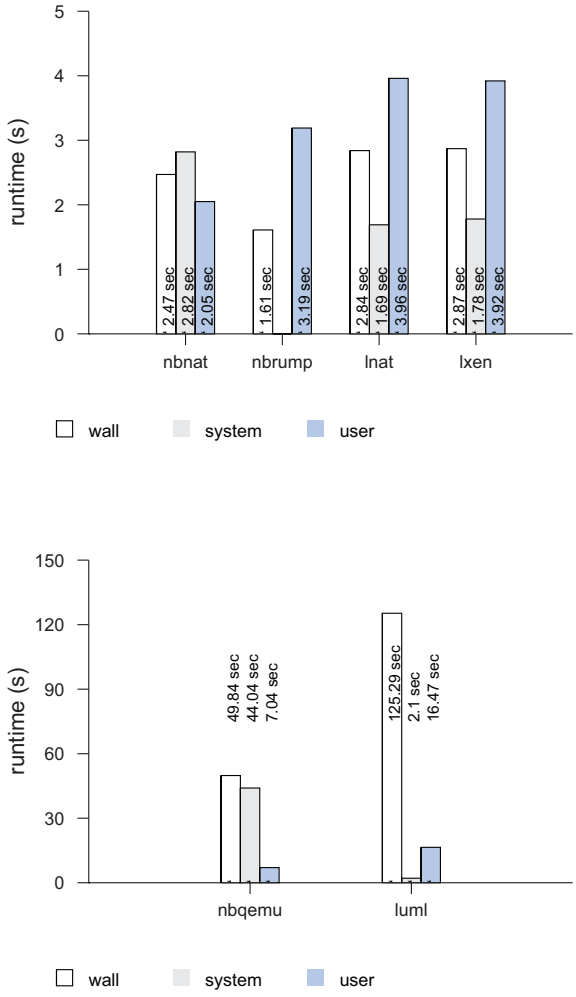

Figure 4.14: **Time to execute 5M system calls per thread in 2 parallel threads.** We divide the measurements into two figures since the durations are vastly different. The prefixes "nb" and "l" denote NetBSD and Linux hosts, respectively. As can be seen by comparing the user/system and walls times in the first figure, the technologies measured there are capable of using more than one host CPU. Smaller wall times are better.

Rump kernel calls, as expected, carry the least overhead and are the faster by over 50% when compared to native system calls. When compared with the UML normalized performance, a rump kernel system call performs 6707% better. We are unsure why UML performance is this poor in wall time even though based on literature [11, 61] we expected it to be significantly slower than a rump kernel. QEMU performance is where we expect it to be.

4.6.4 Networking Latency

To test packet transmission performance in virtual network cluster, we used a linear setup like the one described in Section 4.6.2 and measure the time it takes for a UDP packet to travel from one peer to another and back. The results as a function of the number of hops are displayed in Figure 4.15. In the case of 255 nodes, the RTT translates to a 15.5μs processing time per hop.

The cluster size we tested is limited by the maximum number of hops that the IP time-to-live (TTL) field supports (255). The recommended default from RFC1340 is 64 hops, so we had to adjust the TTL to 255 manually (this adjustment was not an issue in Section 4.6.2, since the ping utility does so automatically).

4.6.5 Backend: Disk File Systems

In the original implementation, the Fast File System (FFS) [69] maintains its consistency by executing critical metadata writes synchronously and improves its performance by allowing non-critical writes (such as writes which do not include an explicit synchronization request) to land on disk asynchronously [69, 71]. The synchronous metadata operations make sure the file system remains consistent on-disk, but especially creating a large number of small files and directories is a slow operation due

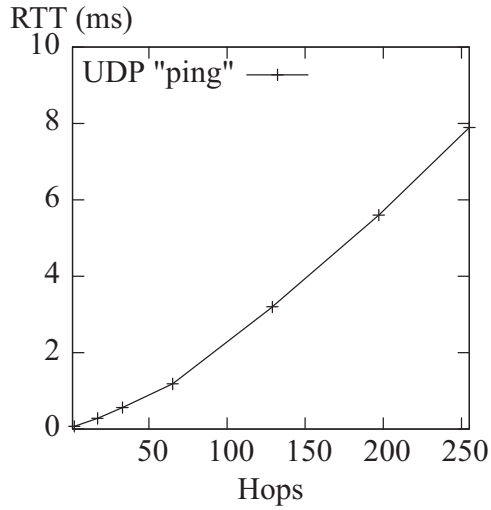

Figure 4.15: **UDP packet RTT.** All rump kernel TCP/IP stacks are run on a single host.

to the number of synchronous metadata writes required. Since then, a variety of techniques have been developed to address this slowness. These techniques include soft updates [72], metadata journaling [85] and a combination of the two previous ones [74].

While the kernel has direct access to the disk driver and can choose to wait for the completion of every write or none at all, the interfaces available to a rump kernel in a POSIX environment are not as fine-grained as ones available in the kernel. We investigated how this affects performance. First, we tested FFS in the traditional mode. These results have been published earlier [55]. Second, we tested how a journaled FFS [85] performs. The test setup is not 100% equivalent, although mostly similar.

Traditional FFS

We measure the performance of three macro level operations: directory traversal with `ls -lR`, recursively copying a directory hierarchy containing both small and large files with `cp -R`, and copying a large file with `cp`. For measuring rump kernel performance we used the *fs-utils* (Section 4.2.1) counterparts of the commands. For the copy operations the source data was precached. The figures are the duration from mount to operation to unmount. Unmounting the file system at the end ensures all caches have been flushed.

We performed the measurements on a 4GB FFS disk image hosted on a regular file and a 20GB FFS partition directly on the hard disk. Both file systems were aged [103]: the first one artificially by copying and deleting files. The latter one has been daily use on the author's laptop for years and has, by definition, aged. The file systems were always mounted so that I/O is performed in the classic manner, i.e. FFS integrity is maintained by performing key metadata operations synchronously. This mode exacerbates the issues with a mix of async and sync I/O requests.

The results are presents in Figure 4.16 and Figure 4.17. The figures between the graphs are not directly comparable, as the file systems have a different layout and different aging. The CD image used for the large copy and the kernel source tree used for the treecopy are the same. The file systems have different contents, so the listing figures are not comparable at all.

Analysis. The results are in line with the expectations.

- The directory traversal shows that the read operations in a rump kernel perform roughly the same on a regular file and 6% slower for an unbuffered backend. This difference is explained by the fact that the buffered file includes read ahead, while the kernel mount accesses the disk unbuffered.

- Copying the large file was measured to consist of 98.5% asynchronous data writes. Memory mapped I/O is almost twice as slow as read/write, since as explained in Section 3.9.2, the relevant parts of the image must be paged in before they can be overwritten and thus I/O bandwidth requirement is double. Unbuffered userspace read/write is 1.5% slower than the kernel mount.

- Copying a directory tree is a mix of directory metadata and file data operations and one third of the I/O is done synchronously in this case. The memory mapped case does not suffer as badly as the large copy, as locality is better. The rump read/write case performs 10% better than the kernel due to a buffered backend. The tradeoff is increased memory use. In the unbuffered case the problem of not being able to issue synchronous write operations while an asynchronous one is in progress shows.

Notably, we did not look into modifying the host kernel to provide more finegrained interfaces for selective cache flushing and I/O to character devices. For now, we maintain that performance for the typical workload is acceptable when compared to a kernel mount. We still emphasize that due to the anykernel a kernel mount can be used for better performance where it is safe to do so.

Journaled FFS

We measured the large file copy and copy of a directory structure tests similarly as for traditional FFS. We did not measure read-only operation since journaling is meant to speed up write operations. We did not measure the MMIO rump kernel block device or the character device backend, since they were already shown to be inferior. The tests were done as in the previous section, with the exception that the file system mounted with `-o log`. The results are presented in Figure 4.18.

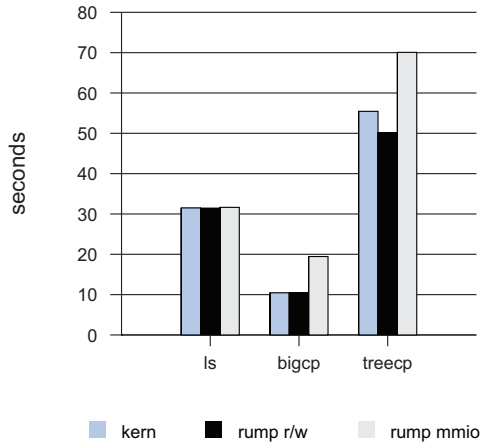

Figure 4.16: **Performance of FFS with the file system a regular file.**

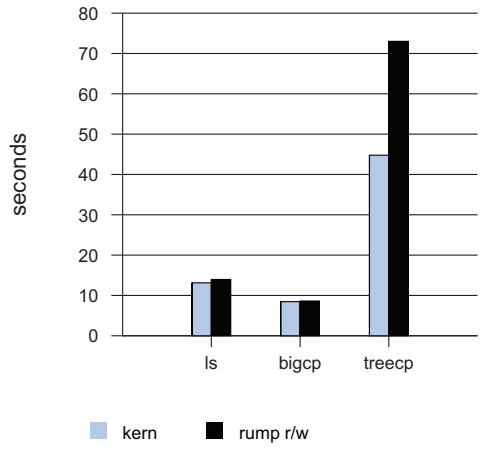

Figure 4.17: **Performance of FFS on a HD partition (raw device).**

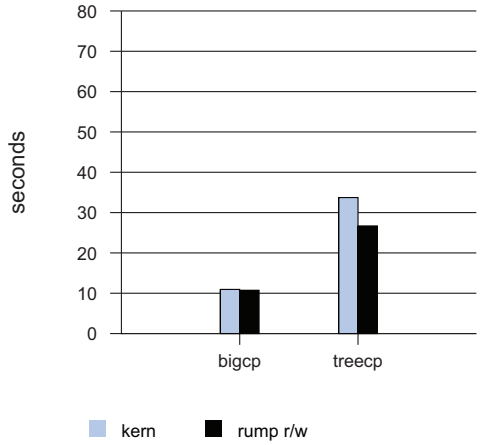

Figure 4.18: Performance of a journaled FFS with the file system on a regular file.

As expected, journaling does not affect writing one large file, since a majority of the operations involve data and not metadata (earlier, we quoted the figure 98.5% of the operations). In the directory write test a rump kernel still outperforms the regular kernel. As above, we conclude this is due to better prefaulting of metadata in the rump kernel backend.

4.6.6 Backend: Networking

We measured the latency of ICMP pinging the TCP/IP stack on common virtualization technologies. The ping response is handled by the TCP/IP stack in the kernel, so there is no process scheduling involved. Since User Mode Linux requires a Linux host, the test against UML was run on Linux. We ran the Xen test with Linux dom0/domU as well.

The results are presented in Figure 4.19. Rump kernels perform second best in the test after User Mode Linux. However, the figures are not directly comparable due to different hosts.

All virtualization technologies perform significantly worse than the native case. This performance drop is because pinging a local address allows the system to route the packet through the loopback address. Since a virtualized OS does not have a local address configured on the host, the packet must be transmitted to the virtual OS instance and back.

4.6.7 Web Servers

A web server performance test provides a macro benchmark of the performance of the TCP/IP stack in a rump kernel. We tested by adapting the thttpd [7] web server as a local client for a rump kernel. We used ApacheBench to measure the total execution time for 10,000 requests of an 80 byte root document on the web server. ApacheBench is always running against the host kernel TCP/IP stack, while the web server was run both against the host stack and the virtual stack. The rump kernel uses the *virt* interface, so as to access the host network. The results are displayed in Figure 4.20.

With concurrency of 4 and above, the difference is about 0.1s in total time. The figure translates to a 0.01ms (3%) difference per request. We attribute this difference to the fact that in addition to the normal interface path, the rump kernel setup must deliver packets through the tap and bridge drivers. We did not attempt to optimize the host for Ethernet access to be available from userspace more directly.

While the above is most likely an obvious result, there are more delicate implications. Running ApacheBench put 10,000 connections in `TIME_WAIT` on the server. This

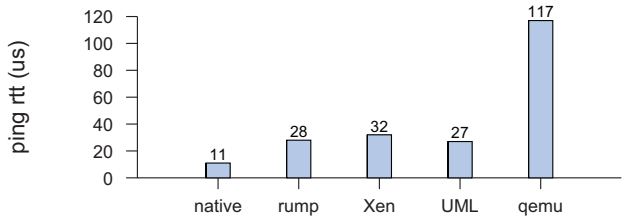

Figure 4.19: **RTT of ping with various virtualization technologies.** The test measures how fast the virtualized kernel responds to an ICMP ping sent from the host. The UML test is not fully comparable, as it was run with Linux as the host operating system.

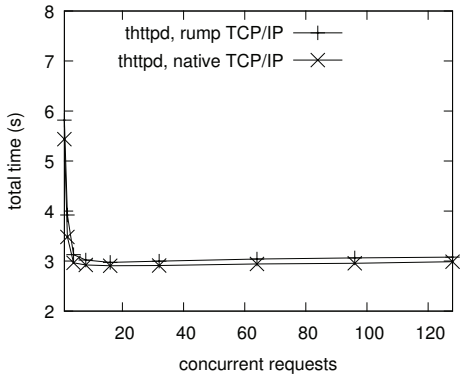

Figure 4.20: **Speed of 10,000 HTTP GET requests over LAN.**

behavior is required by the active close of the TCP state machine. Having 10k connections waiting forced us to wait for the timeout between running the tests for the host kernel networking stack. In contrast, we could kill the process hosting the rump kernel networking stack and start out with a new IP and a clean state in a fraction of a second. While using the host TCP/IP stack was 0.03 times faster than using a TCP/IP stack in the rump kernel, executing the benchmarks on the host stack took over 10 times as long in wall time.

4.7 Summary

We evaluated the anykernel architecture and rump kernels in numerous different ways, both with synthetic benchmarks and analyzing real world data from NetBSD collected between 2007 and 2011.

We found that maintaining the anykernel architecture and rump kernel support in the codebase adds minor maintenance effort. We found that less than 0.2% of repository commits to the kernel source tree caused rump kernel build problems. We added a special *rumptest* build command, which exploits the rump kernel symbol closure to test linking. The command made it 30 times faster to buildtest rump kernel support.

The use of rump kernels as an application library was evaluated with file system related applications and was found to be a working approach. The *makefs* application for NetBSD was reimplemented as a local rump kernel client, and the new version was implemented in under 1/17th of the time taken for the original. This speedup was due to the fact that the existing kernel file system driver could be used directly. The new version also supports four additional file systems because driver support is available without further effort.

We evaluated the portability of the work in two ways. First, we tested running NetBSD rump kernels on foreign platforms. While support for portability is not complete, it works in practice enough to run code. Furthermore, we implemented prototypes for Linux and FreeBSD, and found no reason to suspect that a full implementation would not be possible.

Our security use case demonstrated that file system drivers running inside the kernel are vulnerable to untrusted file system images. A rump kernel running as a micro-kernel style server can be used to isolate vulnerable kernel code into a separate domain when dealing with untrusted images, and retain full in-kernel performance when this isolation is not necessary.

There are close to a thousand tests that use rump kernels in daily NetBSD test runs. We looked at many different types of tests, compared the implementation, result gathering and runtime overhead with other possibilities for testing, and concluded that rump kernels are superior for driver testing. We also included examples of what real-life regressions testing with rump kernels has enabled to detect in NetBSD.

Finally, performance micro benchmarks confirmed that rump kernels are lightweight and fast. The memory overhead for a rump kernel can be as low as 1/20th of that of a full kernel. Bootstrapping a rump kernel is 1,000 times faster than booting a full virtualized kernel in QEMU. We could boot a 255-node networking cluster with 255 virtualized TCP/IP stacks, configure all nodes and run an initial roundtrip packet through them in 1.2 seconds. System call performance for local rump kernels is better than with any other virtualization technology, and was measured to be 6707% faster than for example User-Mode Linux and 50% faster than Xen.

5 Related Work

This chapter surveys and compares work related to the anykernel architecture and rump kernels.

5.1 Running Kernel Code in Userspace

Operating systems running in userspace [26, 31] make it possible to run an entire monolithic operating system inside a userspace process. At a basic level this approach shares the same drawbacks for our purposes as any other full system virtualization solution. The use of the host OS as the VMM can be both a simplifying factor and a complicating one. It simplifies things because no separate VMM is required; the host OS is enough. On the other hand, it complicates things because there is no portability layer. For example, the Dragonfly vkernel [31] relies heavily on host kernel interfaces to be able to create virtual memory spaces for processes of the guest OS. A usermode OS port has overlap with a rump kernel. For example, the virtual I/O drivers described in Section 3.9 can be used equally in a rump kernel and a usermode OS. The differences stem from the fact that while a usermode OS ports the entire OS as-is on top of the host, a rump kernel adapts the OS codebase so that it is both componentized and can relegate functionality directly to the host.

The Alpine [32] network protocol development infrastructure provides an environment for running unmodified FreeBSD 3.3 kernel TCP/IP code in userspace without requiring full virtualization of the source OS. Alpine is implemented before the system call layer by overriding libc with the TCP/IP stack itself run in application process context. It can be viewed as an early and domain specific version of a rump kernel.

Rialto [29] is an operating system with a unified interface both for userspace and the kernel making it possible to run most code in either environment. Rialto was designed and implemented from ground-up as opposed to our approach of starting with an existing system. Interesting ideas include the definition of both internal and external linkage for an interface.

The Linux Kernel Library [98] (LKL) provides the Linux kernel as a monolithic library to be used with 3rd party code such as applications. It is implemented as a separate architecture for Linux which is configured and compiled to produce the library kernel. This approach contrasts the rump kernel approach where the functionality consists of multiple components which are combined by the linker to produce the final result. LKL uses many of the same basic approaches, such as relying on host threads. However, in contrast to our lightweight CPU scheduler, the Linux scheduler is involved with scheduling in LKL, so there are two layers of schedulers at runtime.

5.2 Microkernel Operating Systems

Microkernel operating systems [9, 43, 45, 64] aim for minimal functionality running in privileged mode. The bare minimum is considered to be IPC and scheduling. Driver-level code such as file systems, the networking stack and system call handlers run in unprivileged mode. Microkernel operating systems can be roughly divided into two categories: monolithic and multiserver.

Monolithic servers contain all drivers within a single server. This monolithic nature means that a single component failing will affect the entire server. Monolithic servers can be likened with full system virtualization if the microkernel can be considered a virtual machine monitor. Monolithic servers share the same basic code structure as a monolithic kernel. From an application's perspective the implications are much

the same too: if one component in the monolithic server fails fatally, it may affect all others as well.

Multiserver microkernels divide services into various independent servers. This division means that each component may fail independently. For example, if the TCP/IP server fails, file systems may still be usable (it depends on the application if this single failure has any impact or not).

Part of the effort in building a microkernel operating system is coming up with the microkernel itself and the related auxiliary functions. The rest of the effort is implementing drivers. A rump kernel in itself is agnostic about how it is accessed. As we have shown, it is possible to use rump kernels as microkernel style servers on systems based on the monolithic kernel. The main idea of the anykernel is not to argue how the kernel code should be structured, but rather to give the freedom of running drivers in all configurations.

5.3 Partitioned Operating Systems

Partitioned operating systems [12, 112] run components in independent servers and use message passing for communication between the servers. The motivations for partitioning the OS have to do with the modern processor architecture. The trend is for chips to contain an increasing number of cores, and the structure of the chip the cores are located on starts looking like a networked system with routing and core-to-core latency becoming real issues. Therefore, it makes sense to structure the OS around a networked paradigm with explicit message-passing between servers running on massively multicore architectures. Since rump kernels allow hosting drivers in independent servers, they are a viable option for the driver components of partitioned operating systems.

One argument for partitioned operating systems is that performance and code simplicity is improved since a single parallel thread in a server runs until it blocks. This means that there are no atomic memory bus level locks in the model. We investigated the performance claim in Section 3.5.2 and agree with the performance benefit. However, based on the author's experiences with implementing production quality file servers on top of the puffs [53] cooperative multitasking model, we disagree about concurrent data access being easier in a model without implicit thread scheduling — concurrency still needs to be handled.

5.4 Plan 9

Plan 9 [93] made the idea "everything is a file" reality. Everything in Plan 9 is accessed through the file interface. Kernel devices are driven by *walking* the file name namespace to obtain a handle and then doing read and write operations on the handle. In most cases the control protocol is text, although there are some exceptions for efficiency reasons, such as the frame buffer. Remote resources can be attached to a process's view of the file namespace. The side-effect of this file-oriented operation is that distributed computing is implied; from the application perspective everything looks the same, regardless of whether the resource is local or remote. Since the communication protocol is text, there are no incompatibilities in binary representation between various machine types. However, the practical implication is that the whole system must be built from ground up to conform to the paradigm.

We showed with our remote clients that a Unix system is capable of limited distributed operation without having to rewrite drivers or build the entire OS from ground up. The limitation is that the current system call interface carries many binary subprotocols which are incompatible between heterogeneous hosts. Also, not all standard system calls that applications can expect are supported, `mmap()` being the canonical example — memory mapping cannot be extended to a distributed

environment without the interface contract toward the application being severed. We do not claim that our result is as flexible as the one offered by Plan 9, but we do argue that it offers 98% of the practical functionality with 2% of the effort.

5.5 Namespace Virtualization

Containers

Containers use namespace virtualization to provide multiple different views of the operating system's namespaces to an application, e.g. the file system namespace. These views are provided on top of one kernel instance. Examples of the container approach include FreeBSD jails [52], Linux OpenVZ or the Featherweight Virtual Machine [117]. The effect is the ability to present the illusion of a virtual machine to applications. This illusion is accomplished by keeping track of which namespace each process has access to. When combined with the ability to virtualize the networking stack [118], it is possible to run entire disjoint guest operating systems.

The major difference to rump kernels is that in namespace virtualization all code runs in a single kernel. The implications are reflected in the use cases. Namespace virtualization can be used to present applications a system-defined policy on the namespace they are allowed to view. A rump kernel as a concept is agnostic to who establishes the policy (cf. local and microkernel clients). Namespace virtualization cannot be substituted for any rump kernel use case where the integrity of the kernel is in danger (e.g. testing and secure mounting of disk file systems) since damage to any namespace can affect the entire host kernel and therefore all virtual operating systems offered by it.

View OS

The View OS [37] approach to namespace virtualization is to give the application multiple choices of the entities providing various services conventionally associated with the kernel, e.g. networking or file systems. System calls made by processes are intercepted and redirected to service provides.

System call interception is done using `ptrace()` or alternatively an optimized version called `utrace()`, which requires special host kernel support. This approach allows the redirection policy to exist outside of the client. As servers, the View OS uses specially adapted code which mimics kernel functionality, such as LWIPv6 for the networking service. The clientside functionality and ideology of the View OS and virtualized lightweight servers consisting of preexisting kernel code as offered by rump kernels can be seen to complement each other.

5.6 Lib OS

A library OS means a system where part of the OS functionality runs in the same space as the application. Initially, the library OS was meant to lessen the amount of functionality hidden by the OS abstractions by allowing applications low-level access to OS routines [33, 51]. A later approach [94] used a library as indirection to provide lightweight application sandboxing. What is common between these approaches is that they target the application. We have demonstrated that it is possible, although not most convenient, to use existing kernel code as libraries and access functionality at a mode detailed layer than what is exposed by standard system calls. Our work was focused on the reuse of kernel code, and we did not target application-level sandboxing.

5.7 Inter-OS Kernel Code

The common approach for moving kernel code operating system A to operating system B is to port it. Porting means that the original code from A is taken, incompatible changes are made, and the code is dropped into B. Any improvements made by A or B will not be compatible with B or A, respectively, and if such improvements are desired, they require manual labor.

Another approach is a portability layer. Here the driver code itself can theoretically be shared between multiple systems and differences between systems are handled by a separate portability layer. For example, the USB device drivers in BSD systems used to have a portability header caller `usb_port.h`. Drivers which were written against this header could theoretically be shared between different BSD operating systems. It did not work very well in reality, and use of the macro was retired from NetBSD in 2010. Part of the problem was that there was no established master, and drivers were expected to flow between different operating systems, all of which were continuously evolving.

The portability layer approach is better when the flow of code is unidirectional, i.e. all improvements are made in OS A and at a later date transferred to OS B. For example, the ZFS port of NetBSD has been done this way. New revisions of OpenSolaris could be directly imported into NetBSD without having to merge changes.

Next, we look at various projects which provide 3rd party support with the intent of being able to host the kernel either in or out of the kernel on a foreign system.

OSKit

Instead of making system A's drivers available for system B, the OSKit project [34] had a different motivation. It made system A's drivers available for virtually everyone via portability layers. In fact, there was no single source system, and OSKit offered drivers from multiple different operating systems such as FreeBSD and NetBSD. The approach OSKit took was to take code from the source systems, incorporate it into the OSKit tree, and produce an OSKit release. Since OSKit functionality was not maintained within the source systems, every update takes manual effort. The latest release of OSKit was in 2002.

While the anykernel architecture defines a portability layer for kernel code, we do not claim it is as such the ideal approach for integrating kernel code into foreign monolithic kernels. It does not produce a tightly integrated solution, since many subsystems such as the memory allocator are replicated. However, in a system where drivers are separate servers, such as a multiserver microkernel or a hybrid kernel, the anykernel approach with rump kernel is feasible.

Device Driver Environment (DDE)

The DDE (Device Driver Environment) is set of patches to a guest system which allows hosting drivers on other platforms with the DDEKit interface [1]. These concepts map to reimplemented code in the rump kernel and the rumpuser interface, respectively. The purpose of DDE is to allow to running drivers as servers, and only the lower layers of the kernel are supported. Lower layer operation is comparable to rump kernels with microkernel clients which call into the kernel below the system call layer. As an example of use, it is possible to run the unmodified Linux kernel E1000 PCI NIC driver as userspace server on Linux [111] with DDE/Linux.

NDIS

NDIS stands for "Network Driver Interface Specification". It is the interface against which networking components are implemented on the Windows operating system. Among these components are the network interface card (NIC) drivers, which are classified as *miniport* drivers according to Windows Driver Model. The Windows NIC drivers are interesting for other operating systems for two reasons:

1. There is a large number of different NICs and therefore a large number of NIC drivers are required. Implementing each driver separately for each OS constitutes a large amount of work.
2. Some drivers support devices for which no public documentation is available.

The previous examples of use of foreign kernel code involved having access to the source code. Windows drivers are distributed only as binary and this distribution model places additional limitations. First, the code can only run on the CPU architecture the driver was compiled for, namely i386 and amd64. Second, the system hosting the foreign code must provide the same ABI as the driver — mere API compatibility is not enough.

The relevant part of the Windows kernel ABI, including NDIS, is emulated by the *NDIS wrapper* [36]. The wrapper has been ported to most BSD operating systems, and enables including binary-only Windows NIC drivers. In addition to providing the correct interfaces for the drivers to use, the wrapper must also take care of adjusting the function calling convention, since BSD operating systems use a different one from Windows.

The wrapper implementation redefines all structures used by the NDIS interface. It does not introduce any techniques which could have been useful solutions when

discussing the type incompatibility challenges with running NetBSD code in a foreign host (Section 4.3.1). The fact that is possible to define a function interface to attach drivers suggests that Windows does not suffer from the inline/macro problems which affect our ability to use standard NetBSD kernel modules on non-x86 platforms (Section 3.8.3).

5.8 Safe Virtualized Drivers

Safe drivers in virtualized context depend on the ability to limit driver access to only the necessary resources. In most cases, limiting access is straightforward. For example, file system drivers require access only to the backing storage. In case of malfunction or malice, damage will be limited to everything the driver had access to.

Hardware device drivers are typically more difficult to limit since they perform DMA. For example, PCI devices are bus mastering, and DMA is done by programming the necessary physical memory addresses to the device and letting the device handle the I/O. Incorrect or malicious addresses will be read or written by the device. Limiting DMA access is possible if the hardware includes an IOMMU. This capability has been used for safe unmodified virtual device drivers [62].

Notably though, hardware devices do not imply problems with DMA. We demonstrated working USB hardware device drivers. The rump kernel infrastructure does not currently support hardware device drivers in the general case. Still, there is no reason why it could not be done on systems with IOMMU support.

5.9 Testing and Development

Sun's ZFS file system ships with a userspace testing library, libzpool [8]. In addition to kernel interface emulation routines, it consists of the Data Management Unit and Storage Pool Allocator components of ZFS compiled from the kernel sources. The ztest program plugs directly to these components. This approach has several shortcomings compared to using rump kernels. First, it does not include the entire file system architecture, e.g. the VFS layer. The effort of implementing the VFS interface (in ZFS terms the *ZFS POSIX Layer*) was specifically mentioned as the hardest part of porting ZFS to FreeBSD [22]. Therefore, it should receive non-zero support from the test framework. Second, the approach requires special organization of the code for an individual driver. Finally, the test program is specific to ZFS. In contrast, integrating ZFS into the NetBSD FS-independent rump kernel test framework (described in Section 4.5.3) required roughly 30 minutes of work and 30 lines of code. Using a rump kernel it is possible to test the entire driver stack in userspace from system call to VFS layer to ZFS implementation.

5.10 Single Address Space OS

As the name suggests, a Single Address Space OS (SASOS) runs entirely in a single virtual memory address space, typically 64bit, with different applications located in different parts of the address space. The single system-wide address space is in contrast to the approach where each process runs in its own address space. Protection between applications can be provided either by the traditional memory management hardware approach [17, 41] or by means of software-isolated processes (SIP) [47].

A rump kernel with local clients is essentially a SASOS. It is possible to use multiple process contexts against the local rump kernel with the `rump_lwproc` interfaces. The

difference to the abovementioned SASOSs is that the runtime enforces neither fair scheduling nor protection between these process contexts. In other words, everything is based on cooperation. For our case cooperation is a reasonable assumption, since all processes are provided by the same binary.

Another "single address space" limitation we had to overcome was having the kernel and application in a joint symbol namespace. Any symbols exported both by the application portion and the kernel, e.g. `printf()`, would cause a collision. While there are SASOS solutions which, among other things, handle similar symbol collision problems [23], they require modifications to the linking and loading procedure. Since we wanted things to work out-of-the-box on a Unix-style system, we opted for the renaming approach (Section 3.2.1).

6 Conclusions

We set out with the goals of improving characteristics of monolithic kernel operating systems, namely security, code reusability and testing and development characteristics. Our view is that a codebase's real value lies not in the fact that it exists, but that it has been proven and hardened "out there". Any approach which required us to start building the codebase from scratch for an alternate kernel architecture was not a viable solution for us. In contrast, our work was driven by the needs of the existing codebase, and how to gain maximal use out of it.

We claimed that a properly architected and maintained monolithic kernel provides a flexible codebase. 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 application libraries and network servers, and also in the original monolithic kernel. We showed by implementation that the NetBSD monolithic kernel could be turned into an anykernel with relatively simple modifications.

An anykernel can be instantiated into units which virtualize the minimum support functionality for kernel drivers; support which can be used directly from the host is used directly from the host without a layer of indirection. The virtualized kernel driver instances are called *rump kernels* since they retain only a part of the original features. Our implementation hosts rump kernels in a process on a POSIX host. We discuss other possible hosts at the end of this chapter alongside future work.

The development effort for turning a monolithic kernel into an anykernel is insignificant next to the total effort a real world capable monolithic kernel has received. The anykernel architecture by itself is not a solution to the problems; instead, it is merely a concept which enables solving the problems while still retaining the ability to run the kernel in the original monolithic configuration.

At runtime, a rump kernel can assume the role of an application library or the role of a server. Programs requesting services from rump kernels are called rump kernel clients or simply clients. We defined three client types and implemented support for them. Each client type maps to the best alternative for solving one of our motivating problems.

1. Local: the rump kernel is used in a library capacity. Requests are issues as local function calls.

2. Microkernel: the host routes client requests from regular processes to drivers running in isolated servers.

3. Remote: the client and rump kernel are running in different containers (processes) with the client deciding which services to request from the rump kernel. The kernel and client can exist either on the same host or on different hosts and communicate over the Internet.

Local clients allow using kernel drivers as libraries for applications, where everything runs in a single process. This execution model is not only fast, but also convenient, since to an outside observer it looks just like an application, i.e. it is easy to start, debug and stop one.

Microkernel servers allow running kernel code of questionable stability in a separate address space with total application transparency. However, they require both host support for the callback protocol (e.g. VFS for file systems) and superuser privileges to configure the service (in the case of file systems: mount it).

Remote client system call routing is configured for each client instance separately. However, doing so does not require privileges, and this approach is the best choice for virtual kernels to be used in testing and development.

The key performance characteristics of rump kernels are:

- Bootstrap time until application-readiness is in the order of 10ms. On the hardware we used it was generally under 10ms.
- Memory overhead depends on the components included in the rump kernel, and is typically from 700kB to 1.5MB.
- System call overhead for a local rump kernel is 53% of a native system call for uniprocessor configurations and 44% for a dual CPU configuration (i.e. a rump kernel performs a null system call twice as fast), and less than 1.5% of that of User-Mode Linux (the rump kernel syscall mechanism is over 67 times as fast).

Our case study for improving security was turning a file system mount using an in-kernel driver to one using a rump kernel server. From the user perspective, nothing changes. From the system administrator perspective, the only difference is one extra flag which is required at mount time (`-o rump`). Using isolated servers prevents corrupt and malicious file systems from damaging the host kernel. Due to the anykernel architecture, the ability to use the same file system drivers in kernel mode is still possible.

Our application case study involved a redo of the *makefs* application, which creates a file system image out of a directory tree without relying on in-kernel drivers. Due to the new ability of being able to reuse the kernel file system drivers directly in applications, our reimplementation was done in under 6% of the time the original implementation required. In other words, it required days instead of weeks to implement. For another example of using a rump kernel in an application, see Appendix B.2 where using the in-kernel crypto driver for creating an encrypted disk image is explained using binaries available on an out-of-the-box NetBSD installation.

For testing and development we added close to 1,000 test cases using rump kernels as the test backends. The test suite is run against daily NetBSD changes and has been able to find real regressions, including kernel panics which would have caused the test suite to fail if testing was done against the test host kernel. Testing against rump kernels means that a kernel panic is inconsequential to the test run and allows the test run to complete in constant time irrespective of how many tests cause a kernel crash. Additionally, when problems are discovered, the test program that was used to discover the problem can directly be used to debug the relevant kernel code without the need to set up a separate kernel debugging environment.

6.1 Future Directions and Challenges

The hosting of NetBSD-based rump kernels on other POSIX-style operating systems such as Linux was analyzed. Running a limited set of applications was possible, but interfacing between the host and rump namespaces is not yet supported on a general level. For example, the file status structure is called `struct stat` in the client and rump kernel namespaces, but the binary layouts may not match, since the client uses the host representation and the rump kernel uses the NetBSD representation. If a reference to a mismatching data structure is passed from the client to the rump kernel, the rump kernel will access incorrect fields. Data passed over the namespace boundary need to be translated. NetBSD readily contains system call parameter translation code for a number of operating systems such as Linux and FreeBSD under `sys/compat`. We wish to investigate using this already existing translation code where possible to enlarge the set of supported client ABIs.

Another approach to widening host platform support is to port rump kernels to a completely new type of non-POSIX host. Fundamentally, the host must provide the rump kernel a single memory address space and thread scheduling. Therefore, existing microkernel interfaces such as L4 and Minix make interesting cases. Again,

translation may be needed when passing requests from the clients to NetBSD-based rump kernels. However, since interface use is likely to be limited to a small subset, such as the VFS/vnode interfaces, translation is possible with a small amount of work even if support is not readily provided under `sys/compat`.

Multiserver microkernel systems may want to further partition a rump kernel. We investigated this partitioning with the *sockin* facility. Instead of requiring a full TCP/IP stack in every rump kernel which accesses the network, the sockin facility enables a rump kernel to communicate its intentions to a remote server which does TCP/IP. In our case, the remote server was the host kernel.

The lowest possible target for the rump kernel hypercall layer is firmware and hardware. This adaption would allow the use of anykernel drivers both in bootloaders and lightweight appliances. A typical firmware does not provide a thread scheduler, and this lack would either mandate limited driver support, i.e. running only drivers which do not create or rely on kernel threads, or the addition of a simple thread scheduler in the rump kernel hypervisor. If there is no need to run multiple isolated rump kernels, virtual memory support is not necessary.

An anykernel architecture can be seen as a gateway from current all-purpose operating systems to more specialized operating systems running on ASICs. Anykernels enable the device manufacturer to provide a compact hypervisor and select only the critical drivers from the original OS for their purposes. The unique advantage is that drivers which have been used and proven in general-purpose systems, e.g. the TCP/IP stack, may be included without modification as standalone drivers in embedded products.

References

[1] DDE/DDEKit. URL http://wiki.tudos.org/DDE/DDEKit.

[2] E2fsprogs: Ext2/3/4 Filesystem Utilities. URL http://e2fsprogs.sourceforge.net/.

[3] fuse-ext2. URL http://sourceforge.net/projects/fuse-ext2/.

[4] libguestfs: tools for accessing and modifying virtual machine disk images. URL http://libguestfs.org/.

[5] pkgsrc: The NetBSD Packages Collection. URL http://www.pkgsrc.org/.

[6] Slirp, the PPP/SLIP-on-terminal emulator. URL http://slirp.sourceforge.net/.

[7] thttpd – tiny/turbo/throttling HTTP server. URL http://www.acme.com/software/thttpd/.

[8] ZFS Source Tour. URL http://www.opensolaris.org/os/community/zfs/source/.

[9] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A New Kernel Foundation for UNIX Development. In: Proceedings of the USENIX Summer Technical Conference, pages 93–113.

[10] 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.

[11] 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.

[12] 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.

[13] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pages 41–46.

[14] 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.

[15] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In: Proceedings of the USENIX Summer Technical Conference, pages 87–98.

[16] Jeff Bonwick. Oct 31, 2005. ZFS: The Last Word in Filesystems. URL `http://blogs.oracle.com/bonwick/entry/zfs_the_last_word_in`.

[17] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. 1994. Sharing and Protection in a Single-Address-Space Operating System. ACM Transactions on Computer Systems 12, no. 4, pages 271–307.

[18] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. In: Proceedings of the ACM Symposium on Operating Systems Principles, pages 73–88. ACM.

[19] Adam M. Costello and George Varghese. 1995. Redesigning the BSD Callout and Timer Facilities. Technical Report WUCS-95-23, Washington University.

[20] Charles D. Cranor. 1998. Design and Implementation of the UVM Virtual Memory System. Ph.D. thesis, Washington University.

[21] 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.

[22] Pawel Jakub Dawidek. 2007. Porting the ZFS file system to the FreeBSD operating system. In: Proceedings of AsiaBSDCon, pages 97–103.

[23] Luke Deller and Gernot Heiser. 1999. Linking Programs in a Single Address Space. In: Proceedings of the USENIX Annual Technical Conference, pages 283–294.

[24] Mathieu Desnoyers. 2009. Low-Impact Operating System Tracing. Ph.D. thesis, Ecole Polytechnique de Montréal.

[25] 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.

[26] 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`.

[27] Peter Dinda. January, 2002. The Minet TCP/IP Stack. Technical Report NWU-CS-02-08, Northwestern University Department of Computer Science.

[28] Roland C. Dowdeswell and John Ioannidis. 2003. The CryptoGraphic Disk Driver. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pages 179–186.

[29] Richard Draves and Scott Cutshall. 1997. Unifying the User and Kernel Environments. Technical Report MSR-TR-97-10, Microsoft.

[30] Adam Dunkels. 2001. Design and Implementation of the lwIP TCP/IP Stack. Technical report, Swedish Institute of Computer Science.

[31] Aggelos Economopoulos. 2007. A Peek at the DragonFly Virtual Kernel. LWN.net. URL `http://lwn.net/Articles/228404/`.

[32] David Ely, Stefan Savage, and David Wetherall. 2001. Alpine: A User-Level Infrastructure for Network Protocol Development. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems, pages 171–184.

[33] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In: Proceedings of the ACM Symposium on Operating Systems Principles, pages 251–266.

[34] 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.

[35] 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.

[36] FreeBSD Kernel Interfaces Manual. March 2010. ndis – NDIS miniport driver wrapper.

[37] Ludovico Gardenghi, Michael Goldweber, and Renzo Davoli. 2008. ViewOS: A New Unifying Approach Against the Global View Assumption. In: Proceedings of the 8th International Conference on Computational Science, Part I, pages 287–296.

[38] Tal Garfinkel and Mendel Rosenblum. 2005. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environ-

ments. In: Proceedings of the Workshop on Hot Topics in Operating Systems. URL http://static.usenix.org/event/hotos05/final_papers/full_papers/garfinkel/garfinkel.pdf.

[39] 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.

[40] Andreas Gustafsson. NetBSD-current/i386 build status. URL http://www.gson.org/netbsd/bugs/build/.

[41] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. 1998. The Mungi Single-Address-Space Operating System. Software: Practice and Experience 28, no. 9, pages 901–928.

[42] James P. Hennessy, Damian L. Osisek, and Joseph W. Seigh II. 1989. Passive Serialization in a Multitasking Environment. US Patent 4,809,168.

[43] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. 2006. MINIX 3: a highly reliable, self-repairing operating system. ACM SIGOPS Operating Systems Review 40, no. 3, pages 80–89.

[44] 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.

[45] Dan Hildebrand. 1992. An Architectural Overview of QNX. In: Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, pages 113–126. USENIX Association.

[46] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. 1997. Fault Injection Techniques and Tools. IEEE Computer 30, no. 4, pages 75–82.

[47] Galen C. Hunt and James R. Larus. 2007. Singularity: rethinking the software stack. ACM SIGOPS Operating Systems Review 41, no. 2, pages 37–49.

[48] Xuxian Jiang and Dongyan Xu. 2003. SODA: A Service-On-Demand Architecture for Application Service Hosting Utility Platforms. In: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing, pages 174–183.

[49] Nicolas Joly. May-June 2010. Private communication.

[50] Nicolas Joly. November 2009. Private communication.

[51] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. 1997. Application Performance and Flexibility on Exokernel Systems. In: Proceedings of the ACM Symposium on Operating Systems Principles, pages 52–65.

[52] 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`.

[53] Antti Kantee. 2007. puffs - Pass-to-Userspace Framework File System. In: Proceedings of AsiaBSDCon, pages 29–42.

[54] Antti Kantee. 2009. Environmental Independence: BSD Kernel TCP/IP in Userspace. In: Proceedings of AsiaBSDCon, pages 71–80.

[55] Antti Kantee. 2009. Rump File Systems: Kernel Code Reborn. In: Proceedings of the USENIX Annual Technical Conference, pages 201–214.

[56] Antti Kantee. 2010. Rump Device Drivers: Shine On You Kernel Diamond. In: Proceedings of AsiaBSDCon, pages 75–84.

[57] Brian Kernighan. 2006. Code Testing and Its Role in Teaching. ;login: The USENIX Magazine 31, no. 2, pages 9–18.

[58] 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.

[59] Butler W. Lampson. 1983. Hints for Computer System Design. ACM SIGOPS Operating Systems Review 17, no. 5, pages 33–48.

[60] Greg Lehey. 2006. Debugging kernel problems. URL http://www.lemis.com/grog/Papers/Debug-tutorial/tutorial.pdf.

[61] Ben Leslie, Carl van Schaik, and Gernot Heiser. 2005. Wombat: A Portable User-Mode Linux for Embedded Systems. In: Proceedings of the 6th Linux.Conf.Au. URL http://www.linux.org.au/conf/2005/Papers/Ben%20Leslie/Wombat_%20A%20portable%20user-mode%20Linux%20for%20embedded%20systems/index.html.

[62] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. 2004. Unmodified Device Driver Reuse and Improved System Dependability Via Virtual Machines. In: Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation, pages 17–30.

[63] Jochen Liedtke. 1993. Improving IPC by Kernel Design. In: Proceedings of the ACM Symposium on Operating Systems Principles, pages 175–188.

[64] Jochen Liedtke. 1995. On μ-Kernel Construction. In: Proceedings of the ACM Symposium on Operating Systems Principles, pages 237–250.

[65] 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.

[66] Jim Mauro and Richard McDougall. 2001. Solaris Internals: Core Kernel Architecture. Sun Microsystems, Inc. ISBN 0-13-022496-0.

[67] 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.

[68] 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.

[69] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. 1984. A Fast File System for UNIX. Computer Systems 2, no. 3, pages 181–197.

[70] Marshall Kirk McKusick. September 2007. Implementing FFS. Private communication.

[71] 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.

[72] Marshall Kirk McKusick and Gregory R. Ganger. 1999. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In: Proceedings of the USENIX Annual Technical Conference, pages 1–17.

[73] Marshall Kirk McKusick and Michael J. Karels. 1988. Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel. In: Proceedings of the USENIX Summer Technical Conference, pages 295–304.

[74] Marshall Kirk McKusick and Jeffery Roberson. 2010. Journaled Soft-updates. In: Proceedings of EuroBSDCon 2010. URL `http://www.mckusick.com/BSDCan/bsdcan2010.pdf`.

[75] Julio Merino. Automated Testing Framework. URL `http://www.NetBSD.org/~jmmv/atf/`.

[76] Luke Mewburn. April 2009. Private communication.

[77] Luke Mewburn and Matthew Green. 2003. build.sh: Cross-building NetBSD. In: Proceedings of the USENIX BSD Conference, pages 47–56.

[78] 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.

[79] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. 1988. Kerberos Authentication and Authorization System. In: Project Athena Technical Plan.

[80] 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.

[81] 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.

[82] Madanlal Musuvathi and Dawson R. Engler. 2004. Model Checking Large Network Protocol Implementations. In: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, pages 155–168.

[83] Mutt E-Mail Client. URL `http://www.mutt.org/`.

[84] NetBSD Kernel Interfaces Manual. November 2007. pud – Pass-to-Userspace Device.

[85] NetBSD Kernel Interfaces Manual. November 2010. WAPBL – Write Ahead Physical Block Logging file system journaling.

[86] NetBSD Kernel Interfaces Manual. October 2003. carp – Common Address Redundancy Protocol.

[87] NetBSD Project. URL http://www.NetBSD.org/.

[88] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 89–100.

[89] David Niemi and Alain Knaff. 2007. Mtools. URL http://mtools.linux.lu/.

[90] OpenSSH. URL http://www.openssh.com/.

[91] Thomas J. Ostrand and Elaine J. Weyuker. 2002. The Distribution of Faults in a Large Industrial Software System. SIGSOFT Softw. Eng. Notes 27, pages 55–64.

[92] Rob Pike. 2000. Systems Software Research is Irrelevant. URL http://doc.cat-v.org/bell_labs/utah2000/.

[93] 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.

[94] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, pages 291–304.

[95] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON File Systems. ACM SIGOPS Operating Systems Review 39, no. 5, pages 206–220.

[96] Prashant Pradhan, Srikanth Kandula, Wen Xu, Anees Shaikh, and Erich Nahum. 2002. Daytona : A User-Level TCP Stack. URL http://nms.csail.mit.edu/%7Ekandula/data/daytona.pdf.

[97] The Transport Layer Security (TLS) Protocol. 2008. RFC 5246.

[98] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. 2010. LKL: The Linux Kernel Library. In: Proceedings of the RoEduNet International Conference, pages 328–333.

[99] 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.

[100] Margo I. Seltzer, Keith Bostic, Marshall K. McKusick, and Carl Staelin. 1993. An Implementation of a Log-Structured File System for UNIX. In: Proceedings of the USENIX Winter Technical Conference, pages 307–326.

[101] 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.

[102] A NONSTANDARD FOR TRANSMISSION OF IP DATAGRAMS OVER SERIAL LINES: SLIP. 1988. RFC 1055.

[103] Keith A. Smith and Margo I. Seltzer. 1997. File System Aging—Increasing the Relevance of File System Benchmarks. SIGMETRICS Perform. Eval. Rev. 25, no. 1, pages 203–213.

[104] 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.

[105] Jack Sun, Daniel Fryer, Ashvin Goel, and Angela Demke Brown. 2011. Using Declarative Invariants for Protecting File-System Integrity. In: Proceedings

of the 6th Workshop on Programming Languages and Operating Systems, pages 6:1–6:5.

[106] Miklós Szeredi. FUSE: Filesystem in Userspace. URL http://fuse.sourceforge.net/.

[107] Jason Thorpe. 1998. A Machine-Independent DMA Framework for NetBSD. In: Proceedings of the USENIX Annual Technical Conference, FREENIX track, pages 1–12.

[108] Chris Torek. December 1992. Device Configuration in 4.4BSD. URL http://www.netbsd.org/docs/kernel/config-torek.ps.

[109] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. ACM SIGOPS Operating Systems Review 36, pages 181–194.

[110] 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.

[111] Hannes Weisbach, Björn Döbel, and Adam Lackorzynski. 2011. Generic User-Level PCI Drivers. In: Proceedings of the 13th Real-Time Linux Workshop. URL http://lwn.net/images/conf/rtlws-2011/proc/Doebel.pdf.

[112] David Wentzlaff and Anant Agarwal. 2009. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. ACM SIGOPS Operating Systems Review 43, no. 2, pages 76–85.

[113] David Woodhouse. 2001. JFFS: The Journalling Flash File System. In: Proceedings of the Ottawa Linux Symposium. URL http://www.linuxsymposium.org/archives/OLS/Reprints-2001/woodhouse.pdf.

[114] Gary R. Wright and W. Richard Stevens. 1995. TCP/IP Illustrated, Volume 2. Addison Wesley. ISBN 0-201-63354-X.

[115] 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.

[116] 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`.

[117] Yang Yu, Fanglu Guo, Susanta Nanda, Lap-chung Lam, and Tzi-cker Chiueh. 2006. A Feather-weight Virtual Machine for Windows Applications. In: Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pages 24–34.

[118] Marko Zec. 2003. Implementing a Clonable Network Stack in the FreeBSD Kernel. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pages 137–150.

Appendix A Manual Pages

NAME

rump.dhcpclient -- simple dhcp client for rump kernels

SYNOPSIS

rump.dhcpclient *ifname*

DESCRIPTION

The **rump.dhcpclient** utility is a very simple DHCP client which can be used to apply networking configuration on one interface in a rump kernel. Unlike full DHCP clients, **rump.dhcpclient** does not store leases or renew expired leases. The reason for this is the typical transient nature of a rump kernel. Additionally, **rump.dhcpclient** does not save DNS resolver information.

After having succesfully configured networking, **rump.dhcpclient** prints out the networking configuration and lease time and exits.

Since **rump.dhcpclient** uses bpf(4) to send and receive raw network packets, the server must include support for bpf and vfs (for opening */dev/bpf*). Otherwise, the following diagnostic message is printed:

```
rump.dhcpclient: bpf: Function not implemented
```

SEE ALSO

rump_server(1), bpf(4)

CAVEATS

There is no easy way to release a lease.

NAME

rump.halt -- halt a rump kernel

SYNOPSIS

rump.halt [**-dhn**]

DESCRIPTION

The **rump.halt** utility exits a rump kernel. The file system cache, if present, is flushed. Since a rump kernel does not control its clients, they are not directly affected by **rump.halt**. However, they will be unable to request further services from the halted rump kernel.

The options are as follows:

-d Create a core dump. The core file is saved according to standard userland program coredump rules, and can be later examined with a debugger.

-h By default the process hosting the rump kernel exits. Using this option shuts down rump kernel activity, but does not cause the hosting process to exit.

-n Do not flush the file system cache. This option should be used with extreme caution. It can be used if a virtual disk or a virtual processor is virtually on fire.

SEE ALSO

rump(3)

HISTORY

The **rump.halt** command appeared in NetBSD 6.0.

CAVEATS

While using **-h** makes it impossible to issue further system calls, it does not necessarily stop all activity in a rump kernel. It is recommended this option is used only for debugging purposes.

NAME

rump_server, **rump_allserver** -- rump kernel server

SYNOPSIS

rump_server [**-s**] [**-c** *ncpu*] [**-d** *drivespec*] [**-l** *library*] [**-m** *module*] *url*

DESCRIPTION

The **rump_server** utility is used to provide a rump kernel service. Clients can use the system calls provided by **rump_server** via *url*.

The difference between **rump_server** and **rump_allserver** is that **rump_server** offers only a minimalistic set of features, while **rump_allserver** provides all rump kernel components which were available when the system was built. At execution time it is possible to load components from the command line as described in the options section.

-c *ncpu*

Configure *ncpu* virtual CPUs on SMP-capable archs. By default, the number of CPUs equals the number of CPUs on the host.

-d *drivespec*

The argument *drivespec* maps a host file in the rump fs namespace. The string *drivespec* must be of comma-separated ‘‘name=value’’ format and must contain the following tokens:

key Block device path in rump namespace. This must be specified according to the rules for a key in rump_etfs(3).

hostpath Host file used for storage. If the file does not exist, it will be created.

size Size of the mapping. Similar to dd(1), this argument accepts a suffix as the multiplier for the number. The special value ‘‘host’’ indicates that the current size of *hostpath* will be used. In this case it is assumed that *hostpath* exists and is a regular file.

OR

disklabel Use a disklabel partition identifier to specify the offset and size of the mapping. *hostpath* must contain an existing and valid disklabel within the first 64k.

The following are optional:

offset Offset of the mapping. The window into *hostpath* therefore is [*offset*, *offset+size*]. In case this parameter is not given, the default value 0 is used.

type The type of file that *key* is exposed as within the rump kernel. The possibilities are ‘‘blk’’, ‘‘chr’’, and ‘‘reg’’ for block device, character device and regular file, respectively. The default is a block device.

Note: the contents of block devices are cached in the rump kernel’s buffer cache. To avoid cache incoherency, it is advisable not to access a file

through the host namespace while it is mapped as a block device in a rump kernel.

In case *hostpath* does not exist, it will be created as a regular file with mode 0644 (plus any restrictions placed by umask). In case *hostpath* is a regular file and is not large enough to accommodate the specified size, it will be extended to the specified size.

-l *library*

Call **dlopen**() on library before initializing the rump kernel. In case *library* provides a kernel module, it will appear as a builtin module in the rump kernel. Any rump component present in *library* will also be initialized.

The argument *library* can contain a full path or a filename, in which case the standard dynamic library search path will be used. Libraries are loaded in the order they are given. Dependencies are not autoloaded, and the order must be specified correctly.

-m *module*

Load and link a kernel module after the rump kernel is initialized. For this to work, the rump kernel must include the vfs faction, since the module is loaded using kernel vfs code (see *EXAMPLES*).

-r *total_ram*

Sets the limit of kernel memory allocatable by the server to *total_ram* as opposed to the default which allows the server to allocate as much memory as the host will give it. This parameter is especially useful for VFS servers, since by default the vir-

tual file system will attempt to consume as much memory as it can, and accessing large files can cause an excessive amount of memory to be used as file system cache.

-s Do not detach from the terminal. By default, **rump_server** detaches from the terminal once the service is running on *url*.

-v Set bootverbose.

After use, **rump_server** can be made to exit using rump.halt(1).

EXAMPLES

Start a server and load the tmpfs file system module, and halt the server immediately afterwards:

```
$ rump_server -lrumpvfs -m /modules/tmpfs.kmod unix://sock
$ env RUMP_SERVER=unix://sock rump.halt
```

Start a server with the one gigabyte host file *dk.img* mapped as the block device */dev/dk* in the rump kernel.

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

Start a server which listens on INADDR_ANY port 3755

```
$ rump_server tcp://0:3755/
```

Start a FFS server with a 16MB kernel memory limit.

```
$ rump_server -lrumpvfs -lrumpfs_ffs -r 16m unix:///tmp/ffs_server
```

SEE ALSO

rump.halt(1), dlopen(3), rump(3), rump_sp(7)

NAME

shmif_dumpbus -- examine shmif bus contents

SYNOPSIS

shmif_dumpbus [**-h**] [**-p** *pcapfile*] *busfile*

DESCRIPTION

The **shmif_dumpbus** utility examines the bus of an shmif(4) Ethernet interface. The most useful feature is converting the bus to the pcap(3) file format for later examination. **shmif_dumpbus** itself is limited to displaying only very basic information about each frame.

shmif_dumpbus accepts the following flags:

-h Print bus header only and skip contents.

-p *pcapfile* Convert bus contents to the pcap(3) format and write the result to *pcapfile*. The file - signifies stdout.

EXAMPLES

Feed the busfile contents to pcap:

```
$ shmif_dumpbus -p - busfile | tcpdump -r -
```

SEE ALSO

pcap(3), shmif(4), tcpdump(8)

CAVEATS

shmif_dumpbus does not lock the busfile and is best used for post-mortem analysis of the bus traffic.

The timestamp for each frame contains the sender's timestamp and may not be monotonically increasing with respect to the frame order in the dump.

NAME

p2k -- puffs to kernel vfs translation library

LIBRARY

p2k Library (libp2k, -lp2k)

SYNOPSIS

#include <rump/p2k.h>

*struct p2k_mount **
p2k_init(*uint32_t puffs_flags*);

void
p2k_cancel(*struct p2k_mount *p2m*, *int error*);

int
p2k_setup_fs(*struct p2k_mount *p2m*, *const char *vfsname*,
*const char *devpath*, *const char *mountpath*, *int mntflags*, *void *arg*,
size_t alen);

p2k_setup_diskfs(*struct p2k_mount *p2m*, *const char *vfsname*,
*const char *devpath*, *int partition*, *const char *mountpath*,
int mntflags, *void *arg*, *size_t alen*);

int
p2k_mainloop(*struct p2k_mount *p2m*);

int
p2k_run_fs(*const char *vfsname*, *const char *devpath*,
*const char *mountpath*, *int mntflags*, *void *arg*, *size_t alen*,
uint32_t puffs_flags);

int
p2k_run_diskfs(*const char *vfsname*, *const char *devpath*, *int partition*, *const char *mountpath*, *int mntflags*, *void *arg*, *size_t alen*, *uint32_t puffs_flags*);

DESCRIPTION

The **p2k** library translates the puffs protocol to the kernel vfs(9) protocol and back again. It can therefore be used to mount and run kernel file system code as a userspace daemon.

Calling the library interface function mounts the file system and, if succesful, starts handling requests. The parameters are handled by **ukfs_mount**() (see ukfs(3)), with the exception that *mountpath* and *puffs_flags* are handled by puffs(3). The "run_fs" variants of the interfaces are provided as a convenience for the common case. They execute all of init, setup and mainloop in one call.

ENVIRONMENT

The following environment variables affect the behaviour of **p2k**. They are useful mostly for debugging purposes. The flags are environment variables because typically the command line arguments to **p2k** utilities are parsed using versions not aware of **p2k** options; for example, the rump_cd9660(8) arguments are really parsed by mount_cd9660(8).

P2K_DEBUG Do not detach from tty and print information about each puffs operation. In case the daemon receives SIGINFO (typically from ctrl-T), it dumps out the status of the mount point. Sending SIGUSR1 causes a dump of all the vnodes (verbose).

P2K_NODETACH Do not detach from tty.

P2K_NOCACHE_PAGE Do not use the puffs page cache.

P2K_NOCACHE_NAME Do not use the puffs name cache.

P2K_NOCACHE Do not use the puffs page or name cache.

P2K_WIZARDUID If set, use the value of the variable to determine the UID of the caller of each operation instead of the actual caller supplied by puffs(3). This can be used for example to simplify modifying an OS installation's root image as a non-root user.

SEE ALSO

puffs(3), rump(3), ukfs(3), rump_cd9660(8), rump_efs(8), rump_ext2fs(8), rump_ffs(8), rump_hfs(8), rump_lfs(8), rump_msdos(8), rump_nfs(8), rump_ntfs(8), rump_smbfs(8), rump_syspuffs(8), rump_sysvbfs(8), rump_tmpfs(8), rump_udf(8)

NAME

rump -- The Rump Anykernel

LIBRARY

rump Library (librump, -lrump)

SYNOPSIS

#include <rump/rump.h>

#include <rump/rump_syscalls.h>

DESCRIPTION

rump is part of the realization of a flexible anykernel architecture for NetBSD. An anykernel architecture enables using kernel code in a number of different kernel models. These models include, but are not limited to, the original monolithic kernel, a microkernel server, or an exokernel style application library. **rump** itself makes it possible to run unmodified kernel components in a regular userspace process. Most of the time "unmodified" means unmodified source code, but some architectures can also execute unmodified kernel module binaries in userspace. Examples of different use models are running file system drivers as userspace servers (see p2k(3)) and being able to write standalone applications which understand file system images.

Regardless of the kernel model used, a rump kernel is a fullfledged kernel with its own virtual namespaces, including a file system hierarchy, CPUs, TCP/UDP ports, device driver attachments and file descriptors. This means that any modification to the system state on the host running the rump kernel will not show up in the rump kernel and vice versa. A rump kernel may also be significantly more lightweight than the host, and might not include include for example file system support at all.

Clients using services provided by rump kernels can exist either in the same process as the rump kernel or in other processes. Local clients access the rump kernel through direct function calls. They also naturally have access to the kernel memory space. This document is geared towards local clients. For more information on remote clients, see rump_sp(7). It is also possible to use unmodified application binaries as remote clients with rumphijack(3).

A rump kernel is bootstrapped by calling **rump_init**(). Before bootstrapping the kernel, it is possible to control its functionality by setting various environment variables:

RUMP_NCPU If set, indicates the number of virtual CPUs configured into a rump kernel. The default is the number of host CPUs. The number of virtual CPUs controls how many threads can enter the rump kernel simultaneously.

RUMP_VERBOSE If set to non-zero, activates bootverbose.

RUMP_THREADS If set to 0, prevents the rump kernel from creating any kernel threads. This is possible usually only for file systems, as other subsystems depend on threads to work.

RUMP_MEMLIMIT If set, indicates how many bytes of memory a rump kernel will allocate before attempting to purge caches. The default is as much as the host allows.

RUMP_NVNODES Sets the value of the kern.maxvnodes sysctl node to the indicated amount. Adjusting this may be useful for example when testing vnode reclaim code paths. While the same value can be set by means of sysctl, the env

variable is often more convenient for quick testing. As expected, this option has effect only in rump kernels which support VFS. The current default is 1024 vnodes.

A number of interfaces are available for requesting services from a rump kernel. The most commonly used ones are the rump system calls. They are exactly like regular system calls but with the exception that they target the rump kernel of the current process instead of the host kernel. For example, **rump_sys_socket**() takes the same parameters as **socket**() and will open a socket in the rump kernel. The resulting file descriptor may be used only in other rump system calls and will have undefined results if passed to the host kernel.

Another set of interfaces specifically crafted for rump kernels are the rump public calls. These calls reside in the rump_pub namespace. An example is **rump_pub_module_init**() which initializes a prelinked kernel module.

A rump kernel is constructed at build time by linking a set of libraries with application level code. The mandatory libraries are the kernel base (librump) and the rump hypercall library (librumpuser) which a rump kernel uses to request services from the host. Beyond that, there are three factions which define the flavour of a rump kernel (librumpdev, librumpnet and librumpvfs) and driver components which use features provided by the base and factions. Notably, components may have interdependencies. For example, a rump kernel providing a virtual IP router requires the following components: rumpnet_netinet, rumpnet_net, rumpnet, rumpnet_virtif, rump, and rumpuser. A rump kernel providing an NFS client requires the above and additionally rumpfs_nfs and rumpvfs.

In addition to defining the configuration at link time, it is also possi-

ble to load components at runtime. There are two ways of doing this: using **dlopen**() to link a shared library into a rump kernel and initializing with **rump_pub_module_init**() or specifying a module on the file system to **rump_sys_modctl**() and letting the rump kernel do the linking. Notably, in the latter case debugging with symbols is not possible since the host gdb does not know about symbols loaded by the rump kernel. Generally speaking, dynamically loadable components must follow kernel module boundaries.

SEE ALSO

rump_server(1), p2k(3), rump_etfs(3), rump_lwproc(3), rumpclient(3), rumphijack(3), rumpuser(3), ukfs(3), rump_sp(7)

Antti Kantee, "Environmental Independence: BSD Kernel TCP/IP in Userspace", *Proceedings of AsiaBSDCon 2009*, pp. 71-80, March 2009.

Antti Kantee, "Kernel Development in Userspace - The Rump Approach", *BSDCan 2009*, May 2009.

Antti Kantee, "Rump File Systems: Kernel Code Reborn", *Proceedings of the 2009 USENIX Annual Technical Conference*, pp. 201-214, June 2009.

Arnaud Ysmal and Antti Kantee, "Fs-utils: File Systems Access Tools for Userland", *EuroBSDCon 2009*, September 2009.

Antti Kantee, "Rump Device Drivers: Shine On You Kernel Diamond", *Proceedings of AsiaBSDCon 2010*, pp. 75-84, March 2010.

HISTORY

rump appeared as an experimental concept in NetBSD 5.0. The first stable version was released in NetBSD 6.0.

AUTHORS

Antti Kantee <pooka@iki.fi>

NAME

rump_etfs -- rump host file system interface

LIBRARY

rump kernel (librump, -lrump)

SYNOPSIS

#include <rump/rump.h>

int
rump_pub_etfs_register(*const char *key*, *const char *hostpath*,
enum rump_etfs_type ftype);

int
rump_pub_etfs_register_withsize(*const char *key*, *const char *hostpath*,
enum rump_etfs_type ftype, *uint64_t begin*, *uint64_t size*);

int
rump_pub_etfs_remove(*const char *key*);

DESCRIPTION

The rump ExtraTerrestrial File System (**rump_etfs**) is used to provide access to the host file system namespace within a rump kernel.

The operation is based on registered *key* values which each map to a *hostpath*. A key must be an absolute path (i.e. begin with ‘‘/’’). Multiple leading slashes are collapsed to one (i.e. ‘‘/key’’ is the same as ‘‘//key’’). The rest of the path, including slashes, is compared verbatim (i.e. ‘‘/key/path’’ does not match ‘‘/key//path’’).

The *hostpath* is interpreted in host system context for the current work-

ing directory and can be either absolute or relative.

The *ftype* parameter specifies how etfs file will be presented and does not have to match the host type, although some limitations apply. Possible values are:

RUMP_ETFS_REG — regular file.

RUMP_ETFS_BLK — block device. This is often used when mapping file system images.

RUMP_ETFS_CHR — character device.

RUMP_ETFS_DIR — directory. This option is valid only when *hostpath* is a directory. The immediate children of the host directory will be accessible inside a rump kernel.

RUMP_ETFS_DIR_SUBDIRS — directory. This option is valid only when *hostpath* is a directory. This option recursively applies to all subdirectories, and allows a rump kernel to access an entire directory tree.

The interfaces are:

rump_pub_etfs_register(*key*, *hostpath*, *ftype*)
Map *key* to a file of type *ftype* with the contents of *hostpath*.

rump_pub_etfs_register_withsize(*key*, *hostpath*, *ftype*, *begin*, *size*)
Like the above, but map only [*begin*, *begin+size*] from *hostpath*.

This is useful when mapping disk images where only one partition is relevant to the application. If *size* is given the special value RUMP_ETFS_SIZE_ENDOFF, the underlying file is mapped from *begin* to the end of the file.

rump_pub_etfs_remove(*key*)
Remove etfs mapping for *key*. This routine may be called only if the file related to the mapping is not in use.

EXAMPLES

Map a host image file to a mountable */dev/harddisk* path using window offsets from the disklabel.

```
rump_pub_etfs_register_withsize("/dev/harddisk", "disk.img",
    RUMP_ETFS_BLK,
    pp->p_offset << DEV_BSHIFT, pp->p_size << DEV_BSHIFT);
```

Make the host kernel module directory hierarchy available within the rump kernel.

```
rump_pub_etfs_register("/stand/i386/5.99.41",
    "/stand/i386/5.99.41", RUMP_ETFS_DIR_SUBDIRS);
```

SEE ALSO

rump(3)

HISTORY

rump_etfs first appeared in NetBSD 6.0.

NAME

rump_lwproc -- rump process/lwp management

LIBRARY

rump kernel (librump, -lrump)

SYNOPSIS

#include <rump/rump.h>

int
rump_pub_lwproc_rfork(*int flags*);

int
rump_pub_lwproc_newlwp(*pid_t pid*);

void
rump_pub_lwproc_switch(*struct lwp *l*);

void
rump_pub_lwproc_releaselwp();

*struct lwp **
rump_pub_lwproc_curlwp();

DESCRIPTION

In a normal operating system model a process is a resource container and a thread (lwp) is the execution context. Every lwp is associated with exactly one process, and a process is associated with one or more lwps. The current lwp (curlwp) indicates the current process and determines which resources, such as UID/GID, current working directory, and file descriptor table, are currently used. These basic principles apply to

rump kernels as well, but since rump uses the host's thread and process context directly, the rules for how thread context is determined are different.

In the rump model, each host thread (pthread) is either bound to a rump kernel lwp or accesses the rump kernel with an implicit thread context associated with pid 1. An implicit thread context is created every time the rump kernel is entered and disbanded upon exit. While convenient for occasional calls, creating an implicit thread uses a shared resource which can become highly contended in a multithreaded situation. It is therefore recommended that dedicated threads are created.

The association between host threads and the rump kernel curlwp is left to the caller. It is possible to create a dedicated host thread for every rump kernel lwp or multiplex them on top of a single host thread. After rump lwps have been created, switching curlwp is very cheap -- faster than a thread context switch on the host. In case multiple lwps/processes are created, it is the caller's responsibility to keep track of them and release them when they are no longer necessary. Like other rump kernel resources, procs/lwps will be released when the process hosting the rump kernel exits.

rump_pub_lwproc_rfork()

Create a process, one lwp inside it and set curlwp to the new lwp. The *flags* parameter controls how file descriptors are inherited from the parent. By default (flags=0) file descriptors are shared. Other options are:

RUMP_RFFDG Copy file descriptors from parent. This is what fork(2) does.

RUMP_RFCFDG File descriptors neither copied nor shared, i.e. new process does not have access to the parent's file descriptors.

This routine returns 0 for success or an errno indicating the reason for failure. The new process id can be retrieved in the normal fashion by calling **rump_sys_getpid**().

rump_pub_lwproc_newlwp(*pid*)

Create a new lwp attached to the process specified by *pid*. Sets curlwp to the new lwp. This routine returns 0 for success or an errno indicating the reason for failure.

rump_pub_lwproc_switch(*l*)

Sets curlwp to *l*. In case the new thread is associated with a different process than the current one, the process context is also switched. The special value NULL sets curlwp to implicit context. Switching to an already running lwp, i.e. attempting to use the same curlwp in two host threads simultaneously causes a fatal error.

rump_pub_lwproc_releaselwp()

Release curlwp and set curlwp to context. In case curlwp was the last thread inside the current process, the process container is also released. Calling this routine without a dedicated curlwp is a fatal error.

rump_pub_lwproc_curlwp()

Returns curlwp or NULL if the current context is an implicit context.

SEE ALSO

getpid(2), rump(3)

HISTORY

rump_lwproc first appeared in NetBSD 6.0.

NAME

rumpclient -- rump client library

LIBRARY

library ‘‘rumpclient’’ (librumpclient, -lrumpclient)

SYNOPSIS

#include <rump/rumpclient.h>

#include <rump/rump_syscalls.h>

int
rumpclient_init();

pid_t
rumpclient_fork();

pid_t
rumpclient_vfork();

*struct rumpclient_fork **
rumpclient_prefork();

int
rumpclient_fork_init(*struct rumpclient_fork *rfp*);

void
rumpclient_fork_cancel(*struct rumpclient_fork *rfp*);

int
rumpclient_exec(*const char *path*, *char *const argv[]*,
*char *const envp[]*);

int
rumpclient_daemon(*int nochdir*, *int noclose*);

void
rumpclient_setconnretry(*time_t retrytime*);

int
rumpclient_syscall(*int num*, *const void *sysarg*, *size_t argsize*,
*register_t *retval*);

DESCRIPTION

rumpclient is the clientside implementation of the rump_sp(7) facility. It can be used to connect to a rump kernel server and make system call style requests.

Every connection to a rump kernel server creates a new process context in the rump kernel. By default a process is inherited from init, but through existing connections and the forking facility offered by **rumpclient** it is possible to form process trees.

rumpclient_init()
Initialize **rumpclient**. The server address is determined from the environment variable RUMP_SERVER according to syntax described in rump_sp(7). The new process is registered to the rump kernel with the command name from getprogname(3).

rumpclient_fork()
Fork a rump client process. This also causes a host process fork via fork(2). The child will have a copy of the parent's rump kernel file descriptors.

rumpclient_vfork()

Like above, but the host uses vfork(2).

rumpclient_prefork()

Low-level routine which instructs the rump kernel that the current process is planning to fork. The routine returns a non-NULL cookie if successful.

rumpclient_fork_init(*rfp*)

Low-level routine which works like **rumpclient_init**(), with the exception that it uses the *rfp* context created by a call to **rumpclient_prefork**(). This is typically called from the child of a fork(2) call.

rumpclient_fork_cancel(*rfp*)

Cancel previously initiated prefork context. This is useful for error handling in case a full fork could not be carried through.

rumpclient_exec(*path*, *argv*, *envp*)

This call is a **rumpclient** wrapper around execve(2). The wrapper makes sure that the rump kernel process context stays the same in the newly executed program. This means that the rump kernel PID remains the same and the same rump file descriptors are available (apart from ones which were marked with FD_CLOEXEC).

It should be noted that the newly executed program must call **rumpclient_init**() before any other rump kernel communication can take place. The wrapper cannot do it because it no longer has program control. However, since all rump clients call the init routine, this should not be a problem.

rumpclient_daemon(*noclose*, *nochdir*)

This function performs the equivalent of daemon(3), but also ensures that the internal call to fork(2) is handled properly. This routine is provided for convenience.

rumpclient_setconnretry(*retrytime*)

Set the timeout for how long the client attempts to reconnect to the server in case of a broken connection. After the timeout expires the client will return a failure for that particular request. It is critical to note that after a restablished connection the rump kernel context will be that of a newly connected client. This means all previous kernel state such as file descriptors will be lost. It is largely up to a particular application if this has impact or not. For example, web browsers tend to recover fairly smoothly from a kernel server reconnect, while sshd(8) gets confused if its sockets go missing.

If *retrytime* is a positive integer, it means the number of seconds for which reconnection will be attempted. The value 0 means that reconnection will not be attempted, and all subsequent operations will return the errno ENOTCONN.

Additionally, the following special values are accepted:

RUMPCLIENT_RETRYCONN_INFTIME

Attempt reconnection indefinitely.

RUMPCLIENT_RETRYCONN_ONCE

Attempt reconnect exactly once. What this precisely means depends on the situation: e.g. getting EHOSTUNREACH immedi-

ately or the TCP connection request timeouting are considered to be one retry.

RUMPCLIENT_RETRYCONN_DIE
In case of a broken connection is detected at runtime, call exit(3). This is useful for example in testing. It ensures that clients are killed immediately when they attempt to communicate with a halted server.

rumpclient_syscall(*num*, *sysarg*, *argsize*, *retval*)
Execute an "indirect" system call. In the normal case system calls are executed through the interfaces in <*rump/rump_syscalls.h*> (for example **rump_sys_read**(*fd*, *buf*, *nbytes*)). This interface allows calling the server with pre-marshalled arguments.

Additionally, all of the supported rump system calls are available through this library. See <*rump/rump_syscalls.h*> for a list.

RETURN VALUES

rumpclient routines return -1 in case of error and set errno. In case of success a non-negative integer is returned, where applicable.

SEE ALSO

rump_server(1), rump(3), rump_sp(7)

CAVEATS

Interfaces for a cryptographically authenticated client-server handshake do not currently exist. This can be worked around with e.g. host access control and an ssh tunnel.

NAME

rumphijack -- System call hijack library

LIBRARY

used by ld.so(1)

DESCRIPTION

The ld.so(1) runtime linker can be instructed to load **rumphijack** between the main object and other libraries. This enables **rumphijack** to capture and redirect system call requests to a rump kernel instead of the host kernel.

The behaviour of hijacked applications is affected by the following environment variables:

RUMPHIJACK

If present, this variable specifies which system calls should be hijacked. The string is parsed as a comma-separated list of ‘‘name=value’’ tuples. The possible lefthandside names are:

‘‘path’’ Pathname-based system calls are hijacked if the path the system call is directed to resides under *value*. In case of an absolute pathname argument, a literal prefix comparison is made. In case of a relative pathname, the current working direct is examined. This also implies that neither ‘‘..’’ nor symbolic links will cause the namespace to be switched.

‘‘blanket’’ A colon-separated list of rump path prefixes. This acts almost like ‘‘path’’ with the difference that the prefix does not get removed when passing the path

to the rump kernel. For example, if ‘‘path’’ is */rump*, accessing */rump/dev/bpf* will cause */dev/bpf* to be accessed in the rump kernel. In contrast, if ‘‘blanket’’ contains */dev/bpf*, accessing */dev/bpf* will cause an access to */dev/bpf* in the rump kernel.

In case the current working directory is changed to a blanketed directory, the current working directory will still be reported with the rump prefix, if available. Note, though, that some shells cache the directory and may report something else. In case no rump path prefix has been configured, the raw rump directory is reported.

It is recommended to supply blanketed pathnames as specific as possible, i.e. use */dev/bpf* instead of */dev* unless necessary to do otherwise. Also, note that the blanket prefix does not follow directory borders. In other words, setting the blanket for */dev/bpf* means it is set for *all* pathnames with the given prefix, not just ones in */dev*.

‘‘socket’’ The specifier *value* contains a colon-separated list of which protocol families should be hijacked. The special value ‘‘all’’ can be specified as the first element. It indicates that all protocol families should be hijacked. Some can then be disabled by prepending ‘‘no’’ to the name of the protocol family.

For example, ‘‘inet:inet6’’ specifies that only PF_INET and PF_INET6 sockets should be hijacked,

while ‘‘all:noinet’’ specifies that all protocol families except PF_INET should be hijacked.

‘‘vfs’’ The specifier *value* contains a colon-separated list of which vfs-related system calls should be hijacked. These differ from the pathname-based file system syscalls in that there is no pathname to make the selection based on. Current possible values are ‘‘nfssvc’’, ‘‘getvfsstat’’, and ‘‘fhcalls’’. They indicate hijacking **nfssvc**(), **getvfsstat**(), and all file handle calls, respectively. The file handle calls include **fhopen**(), **fhstat**(), and **fhstatvfs1**().

It is also possible to use ‘‘all’’ and ‘‘no’’ in the same fashion as with the socket hijack specifier.

‘‘sysctl’’ Direct the **__sysctl**() backend of the sysctl(3) facility to the rump kernel. Acceptable values are ‘‘yes’’ and ‘‘no’’, meaning to call the rump or the host kernel, respectively.

‘‘fdoff’’ Adjust the library’s fd offset to the specified value. All rump kernel descriptors have the offset added to them before they are returned to the application. This should be changed only if the application defines a low non-default FD_SETSIZE for **select**() or if it opens a very large number of file descriptors. The default value is 128.

If the environment variable is unset, the default value "path=/rump,socket=all:nolocal" is used. The rationale for this

is to have networked X clients work out-of-the-box: X clients use local sockets to communicate with the server, so local sockets must be used as a host service.

An empty string as a value means no calls are hijacked.

RUMPHIJACK_RETRYCONNECT

Change how rumpclient(3) attempts to reconnect to the server in case the connection is lost. Acceptable values are:

‘‘inftime’’	retry indefinitely
‘‘once’’	retry once, when that connection fails, give up
‘‘die’’	call exit(3) if connection failure is detected
n	Attempt reconnect for n seconds. The value 0 means reconnection is not attempted. The value n must be a positive integer.

See rumpclient(3) for more discussion.

EXAMPLES

Use an alternate TCP/IP stack for firefox with a persistent server connection:

```
$ setenv RUMP_SERVER unix:///tmp/tcpip
$ setenv LD_PRELOAD /usr/lib/librumphijack.so
$ setenv RUMPHIJACK_RETRYCONNECT inftime
$ firefox
```

SEE ALSO

ld.so(1), rump_server(1), rump(3), rumpclient(3), rump_sp(7)

NAME

rumpuser -- rump hypervisor interface

LIBRARY

rump User Library (librumpuser, -lrumpuser)

SYNOPSIS

#include <rump/rumpuser.h>

DESCRIPTION

rumpuser is the hypervisor interface for rump(3) style kernel virtualization. A virtual rump kernel can make calls to the host operating system libraries and kernel (system calls) using **rumpuser** interfaces. Any "slow" hypervisor calls such as file I/O, sychronization wait, or sleep will cause rump to unschedule the calling kernel thread from the virtual CPU and free it for other consumers. When the hypervisor call returns to the kernel, a new scheduling operation takes place.

For example, rump implements kernel threads directly as hypervisor calls to host pthread(3). This avoids the common virtualization drawback of multiple overlapping and possibly conflicting implementations of same functionality in the software stack.

The **rumpuser** interface is still under development and interface documentation is available only in source form from *src/lib/librumpuser*.

SEE ALSO

rump(3)

NAME

ukfs -- user kernel file system library interface

LIBRARY

ukfs Library (libukfs, -lukfs)

SYNOPSIS

#include <rump/ukfs.h>

DESCRIPTION

The **ukfs** library provides direct access to file systems without having to specially mount a file system. Therefore, accessing a file system through **ukfs** requires no special kernel support apart from standard POSIX functionality. As **ukfs** is built upon rump(3), all kernel file systems which are supported by rump are available. It allows to write utilities for accessing file systems without having to duplicate file system internals knowledge already present in kernel file system drivers.

ukfs provides a high-level pathname based interface for accessing file systems. If a lower level interface it desired, rump(3) should be used directly. However, much like system calls, the interfaces of **ukfs**, are self-contained and require no tracking and release of resources. The only exception is the file system handle *struct ukfs* which should be released after use.

INITIALIZATION

int
ukfs_init()

int
ukfs_modload(*const char *fname*)

int
ukfs_modload_dir(*const char *dirname*)

ssize_t
ukfs_vfstypes(*char *buf*, *size_t buflen*)

*struct ukfs **
ukfs_mount(*const char *vfsname*, *const char *devpath*, *const char *mountpath*, *int mntflags*, *void *arg*, *size_t alen*)

*struct ukfs **
ukfs_mount_disk(*const char *vfsname*, *const char *devpath*, *int partition*, *const char *mountpath*, *int mntflags*, *void *arg*, *size_t alen*)

int
ukfs_release(*struct ukfs *ukfs*, *int flags*)

ukfs_init() intializes the library and must be called once per process using **ukfs**.

ukfs_modload() is used at runtime to dynamically load a library which contains a file system module. For this to succeed, the rump(3) library and the module targetted must be compiled with compatible kernel versions and the application must be dynamically linked. Additionally, since this routine does not handle dependencies, all the dependencies of the library must be loaded beforehand. The routine returns -1 for fatal error, 0 for dependency failure and 1 for success.

ukfs_modload_dir() loads all rump(3) file system modules in directory *dirname*. It looks for libraries which begin with *librumpfs_* and end in

.so. The routine tries to handle dependencies by retrying to load libraries which failed due to dependencies. **ukfs_modload_dir**() returns the number of vfs modules loaded or sets errno and returns -1 in case of a fatal error in directory searching. In case a fatal error occurs after some modules have already been loaded, the number of loaded module is returned. Fatal errors in loading the modules themselves are ignored and **ukfs_modload**() should be used directly if finegrained error reporting is desired.

It should be noted that the above routines affect the whole process, not just a specific instance of **ukfs**. It is preferable to call them from only one thread, as the underlying dynamic library interfaces may not be threadsafe.

ukfs_vfstypes() queries the available file system types and returns a nul-terminated list of types separated by spaces in *buf*. The format of the list is equivalent to the one returned by sysctl(3) on the name *vfs.generic.fstypes*. The function returns the length of the string without the trailing nul or -1 for error. Notably, the return value 0 means there are no file systems available. If there is not enough room in the caller's buffer for all file system types, as many as fit will be returned.

ukfs_mount() intializes a file system image. The handle resulting from the operation is passed to all other routines and identifies the instance of the mount analoguous to what a pathname specifies in a normally mounted file system. The parameters are the following:

vfsname
Name of the file system to be used, e.g. MOUNT_FFS.

devpath
Path of file system image. It can be either a regular file, device or, if the file system does not support the concept of a device, an abitrary string, e.g. network address.

mountpath
Path where the file system is mounted to. This parameter is used only by the file system being mounted. Most of the time UKFS_DEFAULTMP is the correct path.

mntflags
Flags as passed to the mount(2) system call, for example MNT_RDONLY. In addition to generic parameters, file system specific parameters such as MNT_LOG (ffs) may be passed here.

arg File system private argument structure. This is passed directly to the file system. It must match what *vfsname* expects.

alen
Size of said structure.

The **ukfs_mount_disk**() function must be used to mount disk-based file systems. It takes the same arguments as **ukfs_mount**(), except for an additional argument signifying the *partition* number. If the image *devpath* contains a disklabel, this value specifies the number of the partition within the image used as the file system backend. If *devpath* does not contain a disklabel, the value UKFS_PARTITION_NONE must be used to signal that the file system backend is the entire image.

ukfs_release() unmounts the file system and releases the resources asso-

ciated with *ukfs*. The return value signals the return value of the unmount operation. If non-zero, *ukfs* will continue to remain valid. The possible values for flags are:

UKFS_RELFLAG_NOUNMOUNT Do not unmount file system, just release ukfs handle. Release always succeeds.

UKFS_RELFLAG_FORCE Forcefully unmount the file system. This means that any busy nodes (due to e.g. **ukfs_chdir**()) will be ignored. Release always succeeds.

OPERATION

int
ukfs_chdir(*struct ukfs *ukfs*, *const char *path*)

int
ukfs_getdents(*struct ukfs *ukfs*, *const char *dirname*, *off_t *off*, *uint8_t *buf*, *size_t bufsize*)

ssize_t
ukfs_read(*struct ukfs *ukfs*, *const char *filename*, *off_t off*, *uint8_t *buf*, *size_t bufsize*)

ssize_t
ukfs_write(*struct ukfs *ukfs*, *const char *filename*, *off_t off*, *uint8_t *buf*, *size_t bufsize*)

int
ukfs_create(*struct ukfs *ukfs*, *const char *filename*, *mode_t mode*)

int
ukfs_mknod(*struct ukfs *ukfs*, *const char *path*, *mode_t mode*, *dev_t dev*)

int
ukfs_mkfifo(*struct ukfs *ukfs*, *const char *path*, *mode_t mode*)

int
ukfs_mkdir(*struct ukfs *ukfs*, *const char *filename*, *mode_t mode*)

int
ukfs_remove(*struct ukfs *ukfs*, *const char *filename*)

int
ukfs_rmdir(*struct ukfs *ukfs*, *const char *filename*)

int
ukfs_link(*struct ukfs *ukfs*, *const char *filename*, *const char *f_create*)

int
ukfs_symlink(*struct ukfs *ukfs*, *const char *filename*, *const char *linkname*)

ssize_t
ukfs_readlink(*struct ukfs *ukfs*, *const char *filename*, *char *linkbuf*, *size_t buflen*)

int
ukfs_rename(*struct ukfs *ukfs*, *const char *from*, *const char *to*)

int
ukfs_stat(*struct ukfs *ukfs*, *const char *filename*, *struct stat*

**file_stat*)

int
ukfs_lstat(*struct ukfs *ukfs*, *const char *filename*, *struct stat *file_stat*)

int
ukfs_chmod(*struct ukfs *ukfs*, *const char *filename*, *mode_t mode*)

int
ukfs_lchmod(*struct ukfs *ukfs*, *const char *filename*, *mode_t mode*)

int
ukfs_chown(*struct ukfs *ukfs*, *const char *filename*, *uid_t uid*, *gid_t gid*)

int
ukfs_lchown(*struct ukfs *ukfs*, *const char *filename*, *uid_t uid*, *gid_t gid*)

int
ukfs_chflags(*struct ukfs *ukfs*, *const char *filename*, *u_long flags*)

int
ukfs_lchflags(*struct ukfs *ukfs*, *const char *filename*, *u_long flags*)

int
ukfs_utimes(*struct ukfs *ukfs*, *const char *filename*, *const struct timeval *tptr*)

int
ukfs_lutimes(*struct ukfs *ukfs*, *const char *filename*, *const struct*

*timeval *tptr*)

The above routines operate like their system call counterparts and the system call manual pages without the ukfs_ prefix should be referred to for further information on the parameters.

The only call which modifies *ukfs* state is **ukfs_chdir**(). It works like chdir(2) in the sense that it affects the interpretation of relative paths. If succesful, all relative pathnames will be resolved starting from the current directory. Currently the call affects all accesses to that particular , but it might be later changed to be thread private.

UTILITIES

int
ukfs_util_builddirs(*struct ukfs *ukfs*, *const char *pathname*, *mode_t mode*)

Builds a directory hierarchy. Unlike mkdir, the *pathname* argument may contain multiple levels of hierarchy. It is not considered an error if any of the directories specified exist already.

SEE ALSO

rump(3)

HISTORY

ukfs first appeared in NetBSD 5.0.

AUTHORS

Antti Kantee <pooka@cs.hut.fi>

NOTES

ukfs should be considered experimental technology and may change without

warning.

BUGS

On Linux, dynamically linked binaries can include support for only one file system due to restrictions with the dynamic linker. If more are desired, they must be loaded at runtime using **ukfs_modload**(). Even though NetBSD does not have this restriction, portable programs should load all file system drivers dynamically.

NAME

shmif -- rump shared memory network interface

SYNOPSIS

#include <rump/rump.h>

int
rump_pub_shmif_create(*const char *path*, *int *ifnum*);

DESCRIPTION

The **shmif** interface uses a memory mapped regular file as a virtual Ethernet bus. All interfaces connected to the same bus see each others' traffic.

Using a memory mapped regular file as a bus has two implications: 1) the bus identifier is not in flat global namespace 2) configuring and using the interface is possible without superuser privileges on the host (normal host file access permissions for the bus hold).

It is not possible to directly access the host networking facilities from a rump virtual kernel using purely **shmif**. However, traffic can be routed to another rump kernel instance which provides both **shmif** and virt(4) networking.

An **shmif** interface can be created in two ways:

- Programmatically by calling **rump_pub_shmif_create**(). The bus pathname is passed in *path*. The number of the newly created interface is available after a succesful call by dereferencing *ifnum*.

- Dynamically at runtime with ifconfig(8) or equivalent using the

create command. In this case the bus path must be configured with ifconfig(8) *linkstr* before the interface address can be configured.

Destroying an **shmif** interface is possible only via ifconfig(8) *destroy*.

SEE ALSO

rump(3), virt(4), ifconfig(8).

NAME

virt -- rump virtual network interface

SYNOPSIS

#include <rump/rump.h>

int
rump_pub_virtif_create(*int num*);

DESCRIPTION

The **virt** interface acts as a link between a rump virtual kernel and a host tap(4) interface. Interface number <n> always corresponds with the host tap interface tap<n>. All data sent by **virt** is written into */dev/tap<n>* and all data read from */dev/tap<n>* is passed as Ethernet input to the rump virtual kernel.

A **virt** interface can be created in two ways:

- Programmatically by calling **rump_pub_virtif_create**().
- Dynamically at runtime with ifconfig(8) or equivalent using the *create* command.

Destroying a **virt** interface is possible only through ifconfig(8) *destroy*.

The host's tap(4) interface can be further bridged with hardware interfaces to provide full internet access to a rump kernel.

SEE ALSO

rump(3), bridge(4), tap(4), brconfig(8), ifconfig(8)

NAME

rump_sp -- rump remote system call support

DESCRIPTION

The **rump_sp** facility allows clients to attach to a rump kernel server over a socket and perform system calls. While making a local rump system call is faster than calling the host kernel, a remote system call over a socket is slower. This facility is therefore meant mostly for operations which are not performance critical, such as configuration of a rump kernel server.

Clients

The NetBSD base system comes with multiple preinstalled clients which can be used to configure a rump kernel and request diagnostic information. These clients run as hybrids partially in the host system and partially against the rump kernel. For example, network-related clients will typically avoid making any file system related system calls against the rump kernel, since it is not guaranteed that a rump network server has file system support. Another example is DNS: since a rump server very rarely has a DNS service configured, host networking is used to do DNS lookups.

Some examples of clients include **rump.ifconfig** which configures interfaces, **rump.sysctl** which is used to access the sysctl(7) namespace and **rump.traceroute** which is used to display a network trace starting from the rump kernel.

Also, almost any unmodified dynamically linked application (for example telnet(1) or ls(1)) can be used as a rump kernel client with the help of system call hijacking. See rumphijack(3) for more information.

Connecting to the server

A remote rump server is specified using an URL. Currently two types of URLs are supported: TCP and local domain sockets. The TCP URL is of the format tcp://ip.address:port/ and the local domain URL is unix://path. The latter can accept relative or absolute paths. Note that absolute paths require three leading slashes.

To preserve the standard usage of the rump clients' counterparts the environment variable RUMP_SERVER is used to specify the server URL. To keep track of which rump kernel the current shell is using, modifying the shell prompt is recommended -- this is analoguous to the visual clue you have when you login from one machine to another.

Client credentials and access control

The current scheme gives all connecting clients root credentials. It is recommended to take precautions which prevent unauthorized access. For a unix domain socket it is enough to prevent access to the socket using file system permissions. For TCP/IP sockets the only available means is to prevent network access to the socket with the use of firewalls. More fine-grained access control based on cryptographic credentials may be implemented at a future date.

EXAMPLES

Get a list of file systems supported by a rump kernel server (in case that particular server does not support file systems, an error will be returned):

```
$ env RUMP_SERVER=unix://sock rump.sysctl vfs.generic.fstypes
```

SEE ALSO

rump_server(1), rump(3), rumpclient(3), rumphijack(3)

HISTORY

rump_sp first appeared in NetBSD 6.0.

Appendix B Tutorial on Distributed Kernel Services

When a rump kernel is coupled with the *sysproxy* facility it is possible to run loosely distributed client-server ”mini-operating systems”. Since there is minimum configuration and the bootstrap time is measured in milliseconds, these environments are very cheap to set up, use, and tear down on-demand.

This section is a tutorial on how to configure and use unmodified NetBSD kernel drivers as userspace services with utilities available from the NetBSD base system. As part of this, it presents various use cases. One uses the kernel cryptographic disk driver (cgd) to encrypt a partition. Another one demonstrates how to operate an FFS server for editing the contents of a file system even though your user account does not have privileges to use the host's `mount()` system call. Additionally, using a userspace TCP/IP server with an unmodified web browser is detailed.

You will need a NetBSD-*current* snapshot from March 31st 2011 or later. Alternatively, although not available when writing this, you can use NetBSD 6 or later.

B.1 Important concepts and a warmup exercise

This section goes over basic concepts which help to understand how to start and use rump servers and clients.

B.1.1 Service location specifiers

A rump kernel service location is specified with an URL. Currently, two types of connections are supported: TCP and local domain sockets (i.e. file system sockets).

TCP connections use standard TCP/IP addressing. The URL is of the format `tcp://ip.address:port/`. A local domain socket binds to a pathname on the local system. The URL format is `unix://socket/path` and accepts both relative and absolute paths. Note that absolute paths require three leading slashes.

Both the client and the server require a service URL to be specified. For the server, the URL designates where the server should listen for incoming connections, and for the client it specifies which server the client should connect to.

B.1.2 Servers

Kernel services are provided by rump servers. Generally speaking, any driver-like kernel functionality can be offered by a rump server. Examples include file systems, networking protocols, the audio subsystem and USB hardware device drivers. A rump server is absolutely standalone and running one does not require for example the creation and maintenance of a root file system.

The program *rump_server* is a component-oriented rump kernel server (manpage available in Appendix A). It can use any combination of available NetBSD kernel components in userspace. In its most basic mode, server offers only bare-bones functionality such as kernel memory allocation and thread support — generally speaking nothing that is alone useful for applications.

Components are specified on the command line using a linker-like syntax. For example, for a server with FFS capability, you need the VFS faction and the FFS component: `rump_server -lrumpvfs -lrumpfs_ffs`. The `-l` option uses the host's `dlopen()` routine to load and link components dynamically. It is also possible to use the NetBSD kernel loader/linker to load ELF objects by supplying `-m` instead, but for simplicity this article always uses `-l`.

The URL the server listens to is supplied as the last argument on the command line and is of the format described in the previous section. Other options, as documented on the manual page, control parameters such as the number of virtual CPUs configured to the rump server and maximum amount of host memory the virtual kernel will allocate.

B.1.3 Clients

Rump clients are programs which interface with the kernel servers. They can either be used to configure the server or act as consumers of the functionality provided by the server. Configuring the IP address for a TCP/IP server is an example of the former, while web browsing is an example of the latter. Clients can be considered to be the userland of a rump kernel, but unlike in a usermode operating system they are not confined to a specific file system setup, and are simply run from the hosting operating system.

A client determines the server it connects from the URL in the `RUMP_SERVER` environment variable.

A client runs as a hybrid in both the host kernel and rump kernel. It uses *essential* functionality from the rump kernel, while all non-essential functionality comes from the host kernel. The direct use of the host's resources for non-essential functionality enables very lightweight services and is what sets rump apart from other forms of virtualization. The set of essential functionality depends on the application. For example, for `ls` fetching a directory listing with `getdents()` is essential functionality, while allocating the memory to which the directory contents are fetched to is non-essential.

The NetBSD base system contains applications which are preconfigured to act as

rump clients. This means that just setting `RUMP_SERVER` will cause these applications to perform their essential functionality on the specified rump kernel server. These applications are distinguished by a ”rump.”-prefix in their command name. The current list of such programs is:

```
rump.cgdconfig  rump.halt       rump.modunload  rump.raidctl    rump.traceroute
rump.dd         rump.ifconfig   rump.netstat    rump.route
rump.dhcpclient rump.modload    rump.ping       rump.sockstat
rump.envstat    rump.modstat    rump.powerd     rump.sysctl
```

Additionally, almost any other dynamically linked binary can act as a rump client, but it is up to the user to specify a correct configuration for *hijacking* the application's essential functionality. Hijacking is demonstrated in later sections of this tutorial.

B.1.4 Client credentials and access control

The current scheme gives all connecting clients root credentials. It is recommended to take precautions which prevent unauthorized access. For a unix domain socket it is enough to prevent access to the socket using file system permissions. For TCP/IP sockets the only available means is to prevent network access to the socket with the use of firewalls. More fine-grained access control based on cryptographic credentials may be implemented at a future date.

B.1.5 Your First Server

Putting everything together, we're ready to start our first rump server. In the following example we start a server, examine the autogenerated hostname it was

given, and halt the server. We also observe that the socket is removed when the server exits.

```
golem> rump_server unix://rumpserver
golem> ls -l rumpserver
srwxr-xr-x  1 pooka  users  0 Mar 11 14:49 rumpserver
golem> sysctl kern.hostname
kern.hostname = golem.localhost
golem> export RUMP_SERVER=unix://rumpserver
golem> rump.sysctl kern.hostname
kern.hostname = rump-06341.golem.localhost.rumpdomain
golem> rump.halt
golem> rump.sysctl kern.hostname
rump.sysctl: prog init failed: No such file or directory
golem> ls -l rumpserver
ls: rumpserver: No such file or directory
```

As an exercise, try the above, but use `rump.halt` with `-d` to produce a core dump. Examine the core with `gdb` and especially look at the various thread that were running (in gdb: `thread apply all bt`). Also, try to create another core with `kill -ABRT`. Notice that you will have a stale socket in the file system when the server is violently killed. You can remove it with `rm`.

As a final exercise, start the server with `-s`. This causes the server to not detach from the console. Then kill it either with `SIGTERM` from another window (the default signal send by `kill`) or by pressing Ctrl-C. You will notice that the server reboots itself cleanly in both cases. If it had file systems, those would be unmounted too. These features are useful for quick iteration when debugging and developing kernel code.

In case you want to use a debugger to further examine later cases we go over in this tutorial, it is recommended you install debugging versions of rump com-

ponents. That can be done simply by going into `src/sys/rump` and running `make DBG=-g cleandir dependall` and after that `make install` as root. You can also install the debugging versions to an alternate directory with the command `make DESTDIR=/my/dir install` and run the code with `LD_LIBRARY_PATH` set to `/my/dir`. This scheme also allows you to run kernel servers with non-standard code modifications on a non-privileged account.

B.2 Userspace cgd encryption

The cryptographic disk driver, cgd, provides an encrypted view of a block device. The implementation is kernel-based. This makes it convenient and efficient to layer the cryptodriver under a file system so that all file system disk access is encrypted. However, using a kernel driver requires that the code is loaded into the kernel and that a user has the appropriate privileges to configure and access the driver.

Occasionally, it is desirable to encrypt a file system image before distribution. Assume you have a USB image, i.e. one that can boot and run directly from USB media. The image can for example be something you created yourself, or even one of the standard USB installation images offered at ftp.NetBSD.org. You also have a directory tree with confidential data you wish to protect with cgd. This example demonstrates how to use a rump cgd server to encrypt your data. This approach, as opposed to using a driver in the host kernel, has the following properties:

- uses out-of-the-box tools on any NetBSD installation
- does not require any special kernel drivers
- does not require superuser access
- is portable to non-NetBSD systems (although requires some amount of work)

While there are multiple steps with a fair number of details, in case you plan on doing this regularly, it is possible to script them and automate the process. It is recommended that you follow these instructions as non-root to avoid accidentally overwriting a cgd partition on your host due to a mistyped command.

Let's start with the USB disk image you have. It will have a disklabel such as the following:

```
golem> disklabel usb.img
# usb.img:
type: unknown
disk: USB image
label:
flags:
bytes/sector: 512
sectors/track: 63
tracks/cylinder: 16
sectors/cylinder: 1008
cylinders: 1040
total sectors: 1048576
rpm: 3600
interleave: 1
trackskew: 0
cylinderskew: 0
headswitch: 0           # microseconds
track-to-track seek: 0  # microseconds
drivedata: 0

16 partitions:
#        size    offset     fstype [fsize bsize cpg/sgs]
 a:    981792        63     4.2BSD   1024  8192     0  # (Cyl.      0*-    974*)
 b:     66721    981855       swap                     # (Cyl.    974*-   1040*)
 c:   1048513        63     unused      0     0        # (Cyl.      0*-   1040*)
 d:   1048576         0     unused      0     0        # (Cyl.      0 -   1040*)
```

Our goal is to add another partition after the existing ones to contain the cgd-encrypted data. This will require extending the file containing the image, and, naturally, a large enough USB mass storage device onto which the new image file can be copied.

First, we create a file system image out of our data directory using the standard `makefs` command from the NetBSD base system:

```
golem> makefs unencrypted.ffs preciousdir
Calculated size of ‘unencrypted.ffs’: 12812288 bytes, 696 inodes
Extent size set to 8192
unencrypted.ffs: 12.2MB (25024 sectors) block size 8192, fragment size 1024
        using 1 cylinder groups of 12.22MB, 1564 blks, 768 inodes.
super-block backups (for fsck -b #) at:
 32,
Populating ‘unencrypted.ffs’
Image ‘unencrypted.ffs’ complete
```

Then, we calculate the image size in disk sectors by dividing the image size with the disk sector size (512 bytes):

```
golem> expr ‘stat -f %z unencrypted.ffs‘ / 512
25024
```

We then edit the existing image label so that there is a spare partition large enough to hold the image. We need to edit ”total sectors”, and the ”c” and ”d” partition. We also need to create the ”e” partition. Make sure you use ”unknown” instead of ”unused” as the fstype for for partition e. In the following image the edited fields are denotated with a red color.

```
golem> disklabel -re usb.img
# usb.img:
type: unknown
disk: USB image
label:
flags:
bytes/sector: 512
sectors/track: 63
tracks/cylinder: 16
sectors/cylinder: 1008
cylinders: 1040
total sectors: 1073600
rpm: 3600
interleave: 1
trackskew: 0
cylinderskew: 0
headswitch: 0           # microseconds
track-to-track seek: 0  # microseconds
drivedata: 0

16 partitions:
#        size    offset     fstype [fsize bsize cpg/sgs]
 a:    981792        63     4.2BSD   1024  8192     0  # (Cyl.      0*-    974*)
 b:     66721    981855       swap                     # (Cyl.    974*-   1040*)
 c:   1073537        63     unused      0     0        # (Cyl.      0*-   1065*)
 d:   1073600         0     unused      0     0        # (Cyl.      0 -   1065*)
 e:     25024   1048576    unknown      0     0        # (Cyl.   1040*-   1065*)
```

Now, it is time to start a rump server for writing the encrypted data to the image. We need to note that a rump kernel has a local file system namespace and therefore cannot in its natural state see files on the host. However, the `-d` parameter to rump_server can be used to map files from the host into the rump kernel file system namespace. We start the server in the following manner:

```
golem> export RUMP_SERVER=unix:///tmp/cgdserv
golem> rump_server -lrumpvfs -lrumpkern_crypto -lrumpdev -lrumpdev_disk \
    -lrumpdev_cgd -d key=/dk,hostpath=usb.img,disklabel=e ${RUMP_SERVER}
```

This maps partition "e" from the disklabel on `usb.img` to the key `/dk` inside the rump kernel. In other words, accessing sector 0 from `/dk` in the rump kernel namespace will access sector 1048576 on `usb.img`. The image file is also automatically extended so that the size is large enough to contain the entire partition.

Note that everyone who has access to the server socket will have root access to the kernel server, and hence the data you are going to encrypt. In case you are following these instructions on a multiuser server, it is a good idea to make sure the socket is in a directory only you have access to (directory mode 0700).

We can now verify that dumping the entire partition gives us a zero-filled partition of the right size (25024 sectors * 512 bytes/sector = 12812288 bytes):

```
golem> rump.dd if=/dk bs=64k > emptypart
195+1 records in
195+1 records out
12812288 bytes transferred in 0.733 secs (17479246 bytes/sec)
golem> hexdump -x emptypart
0000000    0000    0000    0000    0000    0000    0000    0000    0000
*
0c38000
```

In the above example we could pipe `rump.dd` output directly to `hexdump`. However, running two separate commands also conveniently demonstrates that we get the right amount of data from `/dk`.

If we were to `dd` our `unencrypted.img` to `/dk`, we would have added a regular

unencrypted partition to the image. The next step is to configure a cgd so that we can write encrypted data to the partition. In this example we'll use a password-based key, but you are free to use anything that is supported by `cgdconfig`.

```
golem> rump.cgdconfig -g aes-cbc > usb.cgdparams
golem> cat usb.cgdparams
algorithm aes-cbc;
iv-method encblkno1;
keylength 128;
verify_method none;
keygen pkcs5_pbkdf2/sha1 {
        iterations 325176;
        salt AAAAgGc4DWwqXN4t0eapskSLWTs=;
};
```

Note that if you have a fast machine and wish to use the resulting encrypted partition on slower machines, it is a good idea to edit "iterations". The value is automatically calibrated by cgdconfig so that encryption key generation takes about one second on the platform where the params file is generated [28]. Key generation can take significantly longer on slower systems.

The next step is to configure the cgd device using the paramsfile. Since we are using password-based encryption we will be prompted for a password. Enter any password you want to use to access the data later.

```
golem> rump.cgdconfig cgd0 /dk usb.cgdparams
/dk's passphrase:
```

If we repeat the dd test in the encrypted partition we will get a very different result than above. This is expected, since now we have an encrypted view of the zero-filled partition.

```
golem> rump.dd if=/dev/rcgd0d | hexdump -x | sed 8q
0000000    9937    5f33    25e7    c341    3b67    c411    9d73    645c
0000010    5b7c    23f9    b694    e732    ce0a    08e0    9037    2b2a
*
0000200    0862    ee8c    eafe    b21b    c5a3    4381    cdb5    2033
0000210    5b7c    23f9    b694    e732    ce0a    08e0    9037    2b2a
*
0000400    ef06    099d    328d    a35d    f4ab    aac0    6aba    d673
0000410    5b7c    23f9    b694    e732    ce0a    08e0    9037    2b2a
```

NOTE: The normal rules for the raw device names apply, and the correct device path is `/dev/rcgd0c` on non-x86 archs.

To encrypt our image, we simply need to dd it to the cgd partition.

```
golem> dd if=unencrypted.ffs bs=64k | rump.dd of=/dev/rcgd0d bs=64k
195+1 records in
195+1 records out
12812288 bytes transferred in 0.890 secs (14395829 bytes/sec)
195+1 records in
195+1 records out
12812288 bytes transferred in 0.896 secs (14299428 bytes/sec)
```

We have now successfully written an encrypted version of the file system to the image file and can proceed to shut down the rump server. This makes sure all rump kernel caches are flushed.

```
golem> rump.halt
golem> unset RUMP_SERVER
```

You will need to make sure the cgd params file is available on the platform you intend to use the image on. There are multiple ways to do this. It is safe even to

offer the params file for download with the image — just make sure the password is not available for download. Notably, though, you will be telling everyone how the image was encrypted and therefore lose the benefit of two-factor authentication.

In this example, we use fs-utils [116] to copy the params file to the unencrypted ”a” partition of the image. That way, the params files is included with the image. Like other utilities in this tutorial, fs-utils works purely in userspace and does not require special privileges or kernel support.

```
golem> fsu_put usb.img%DISKLABEL:a% usb.cgdparams root/
golem> fsu_ls usb.img%DISKLABEL:a% -l root/usb.cgdparams
-rw-r--r--  1 pooka  users  175 Feb  9 17:50 root/usb.cgdparams
golem> fsu_chown usb.img%DISKLABEL:a% 0:0 root/usb.cgdparams
golem> fsu_ls usb.img%DISKLABEL:a% -l root/usb.cgdparams
-rw-r--r--  1 root  wheel  175 Feb  9 17:50 root/usb.cgdparams
```

Alternatively, we could use the method described later in Section B.4. It works purely with base system utilities.

We are ready to copy the image to a USB stick. This step should be executed with appropriate privileges for raw writes to USB media. If USB access is not possible on the same machine, the image may be copied over network to a suitable machine.

```
golem# dd if=usb.img of=/dev/rsd0d bs=64k
8387+1 records in
8387+1 records out
549683200 bytes transferred in 122.461 secs (4488638 bytes/sec)
```

Finally, we can boot the target machine from the USB stick, configure the encrypted partition, mount the file system, and access the data. Note that to perform these operations we need root privileges on the target machine.

```
demogorgon# cgdconfig cgd0 /dev/sd0e /root/usb.cgdparams
/dev/sd0e's passphrase:
demogorgon# disklabel cgd0
# /dev/rcgd0d:
type: cgd
disk: cgd
label: fictitious
flags:
bytes/sector: 512
sectors/track: 2048
tracks/cylinder: 1
sectors/cylinder: 2048
cylinders: 12
total sectors: 25024
rpm: 3600
interleave: 1
trackskew: 0
cylinderskew: 0
headswitch: 0           # microseconds
track-to-track seek: 0  # microseconds
drivedata: 0

4 partitions:
#        size    offset     fstype [fsize bsize cpg/sgs]
 a:     25024         0     4.2BSD      0     0     0  # (Cyl.      0 -     12*)
 d:     25024         0     unused      0     0        # (Cyl.      0 -     12*)
disklabel: boot block size 0
disklabel: super block size 0
demogorgon# mount /dev/cgd0a /mnt
demogorgon# ls -l /mnt
total 1
drwxr-xr-x  9 1323  users  512 Feb  4 13:13 nethack-3.4.3
#
```

B.3 Networking

This section explains how to run any dynamically linked networking program against a rump TCP/IP stack without requiring any modifications to the application, including no recompilation. The application we use in this example is the Firefox browser. It is an interesting application for multiple reasons. Segregating the web browser to its own TCP/IP stack is an easy way to increase monitoring and control over what kind of connections the web browser makes. It is also an easy way to get some increased privacy protection (assuming the additional TCP/IP stack can have its own external IP). Finally, a web browser is largely ”connectionless”, meaning that once a page has been loaded a TCP/IP connection can be discarded. We use this property to demonstrate killing and restarting the TCP/IP stack from under the application without disrupting the application itself.

A rump server with TCP/IP capability is required. If the plan is to access the Internet, the *virt* interface must be present in the rump kernel and the host kernel must have support for *tap* and *bridge*. You also must have the appropriate privileges for configuring the setup — while rump kernels do not themselves require privileges, they cannot magically access host resources without the appropriate privileges. If you do not want to access the Internet, using the *shmif* interface is enough and no privileges are required. However, for purposes of this tutorial we will assume you want to access the Internet and all examples are written for the virt+tap+bridge combination.

Finally, if there is a desire to configure the rump TCP/IP stack with DHCP, the rump kernel must support *bpf*. Since bpf is accessed via a file system device node, vfs support is required in this case (without bpf there is no need for file system support). Putting everything together, the rump kernel command line looks like this:

```
rump_server -lrumpnet -lrumpnet_net -lrumpnet_netinet   # TCP/IP networking
    -lrumpvfs -lrumpdev -lrumpdev_bpf                   # bpf support
    -lrumpnet_virtif                                    # virt(4)
```

So, to start the TCP/IP server execute the following. Make sure `RUMP_SERVER` stays set in the shell you want to use to access the rump kernel.

```
golem> export RUMP_SERVER=unix:///tmp/netsrv
golem> rump_server -lrumpnet -lrumpnet_net -lrumpnet_netinet -lrumpvfs
    -lrumpdev -lrumpdev_bpf -lrumpnet_virtif $RUMP_SERVER
```

The TCP/IP server is now running and waiting for clients at `RUMP_SERVER`. For applications to be able to use it, we must do what we do to a regular host kernel TCP/IP stack: configure it. This is discussed in the next section.

B.3.1 Configuring the TCP/IP stack

A kernel mode TCP/IP stack typically has access to networking hardware for sending and receiving packets, so first we must make sure the rump TCP/IP server has the same capability. The canonical way is to use bridging and we will present that here. An alternative is to use the host kernel to route the packets, but that is left as an exercise to the reader. In both cases, the rump kernel sends and receives external packets via a `/dev/tap<n>` device node. The rump kernel must have read-write access to this device node. The details are up to you, but the recommended way is to use appropriate group privileges.

To create a tap interface and attach it via bridge to a host Ethernet interface we execute the following commands. You can attach as many tap interfaces to a single

bridge as you like. For example, if you run multiple rump kernels on the same machine, adding all the respective tap interfaces on the same bridge will allow the different kernels to see each others' Ethernet traffic.

Note that the actual interface names will vary depending on your system and which tap interfaces are already in use.

```
golem# ifconfig tap0 create
golem# ifconfig tap0 up
golem# ifconfig tap0
tap0: flags=8943<UP,BROADCAST,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
        address: f2:0b:a4:f1:da:00
        media: Ethernet autoselect
golem# ifconfig bridge0 create
golem# brconfig bridge0 add tap0 add re0
golem# brconfig bridge0 up
golem# brconfig bridge0
bridge0: flags=41<UP,RUNNING>
        Configuration:
                priority 32768 hellotime 2 fwddelay 15 maxage 20
                ipfilter disabled flags 0x0
        Interfaces:
                re0 flags=3<LEARNING,DISCOVER>
                        port 2 priority 128
                tap0 flags=3<LEARNING,DISCOVER>
                        port 4 priority 128
        Address cache (max cache: 100, timeout: 1200):
                b2:0a:53:0b:0e:00 tap0 525 flags=0<>
                go:le:ms:re:0m:ac re0 341 flags=0<>
```

That takes care of support on the host side. The next task is to create an interface within the rump kernel which uses the tap interface we just created. In case you are not using tap0, you need to know that virt<n> corresponds to the host's tap<n>.

```
golem> rump.ifconfig virt0 create
golem> rump.ifconfig virt0
virt0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        address: b2:0a:bb:0b:0e:00
```

In case you do not have permission to open the corresponding tap device on the host, or the host's tap interface has not been created, you will get an error from ifconfig when trying to create the virt interface.

Now the rump kernel interface exists. The final step is to configure an address and routing. In case there is DHCP support on the network you bridged the rump kernel to, you can simply run `rump.dhcpclient`:

```
golem> rump.dhcpclient virt0
virt0: adding IP address 192.168.2.125/24
virt0: adding route to 192.168.2.0/24
virt0: adding default route via 192.168.2.1
lease time: 172800 seconds (2.00 days)
```

If there is no DHCP service available, you can do the same manually with the same result.

```
golem> rump.ifconfig virt0 inet 192.168.2.125 netmask 0xffffff00
golem> rump.route add default 192.168.2.1
add net default: gateway 192.168.2.1
```

You should now have network access via the rump kernel. You can verify this with a simple ping.

```
golem> rump.ping www.NetBSD.org
PING www.NetBSD.org (204.152.190.12): 56 data bytes
64 bytes from 204.152.190.12: icmp_seq=0 ttl=250 time=169.102 ms
64 bytes from 204.152.190.12: icmp_seq=1 ttl=250 time=169.279 ms
64 bytes from 204.152.190.12: icmp_seq=2 ttl=250 time=169.633 ms
^C
----www.NetBSD.org PING Statistics----
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 169.102/169.338/169.633/0.270 ms
```

In case everything is working fine, you will see approximately the same latency as with the host networking stack.

```
golem> ping www.NetBSD.org
PING www.NetBSD.org (204.152.190.12): 56 data bytes
64 bytes from 204.152.190.12: icmp_seq=0 ttl=250 time=169.134 ms
64 bytes from 204.152.190.12: icmp_seq=1 ttl=250 time=169.281 ms
64 bytes from 204.152.190.12: icmp_seq=2 ttl=250 time=169.497 ms
^C
----www.NetBSD.org PING Statistics----
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 169.134/169.304/169.497/0.183 ms
```

B.3.2 Running applications

We are now able to run arbitrary unmodified applications using the TCP/IP stack provided by the rump kernel. We just have to set `LD_PRELOAD` to instruct the dynamic linker to load the rump hijacking library. Also, now is a good time to make sure `RUMP_SERVER` is still set and points to the right place.

```
golem> export LD_PRELOAD=/usr/lib/librumphijack.so
```

Congratulations, that's it. Any application you run from the shell in which you set the variables will use the rump TCP/IP stack. If you wish to use another rump TCP/IP server (which has networking configured), simply adjust `RUMP_SERVER`. Using this method you can for example segregate some "evil" applications to their own networking stack.

B.3.3 Transparent TCP/IP stack restarts

Since the TCP/IP stack is running in a separate process from the client, it is possible kill and restart the TCP/IP stack from under the application without having to restart the application. Potential uses for this are to take features available in later releases into use or fixing a security vulnerability. Even though NetBSD kernel code barely ever crashes, it does happen, and this will also protect against that.

Since networking stack code does not contain any checkpointing support, killing the hosting process will cause all kernel state to go missing and for example previously used sockets will not be available after restart. Even if checkpointing were added for things like file descriptors, generally speaking checkpointing a TCP connection is not possible. The reaction to this unexpected loss of state largely depends on the application. For example, ssh will not handle this well, but Firefox will generally speaking recover without adverse effects.

Before starting the hijacked application you should instruct the rump client library to retry to connect to the server in case the connection is lost. This is done by setting the `RUMPHIJACK_RETRYCONNECT` environment variable.

```
golem> export RUMPHIJACK_RETRYCONNECT=inftime
golem> firefox
```

Now we can use Firefox just like we would with the host kernel networking stack. When we want to restart the TCP/IP stack, we can use any method we'd like for killing the TCP/IP server, even kill -9 or just having it panic. The client will detect the severed connection and print out the following diagnostic warnings.

```
rump_sp: connection to kernel lost, trying to reconnect ...
rump_sp: still trying to reconnect ...
rump_sp: still trying to reconnect ...
```

Once the server has been restarted, the following message will be printed. If the server downtime was long, the client can take up to 10 seconds to retry, so do not be surprised if you do not see it immediately.

```
rump_sp: reconnected!
```

Note that this message only signals that the client has a connection to the server. In case the server has not been configured yet to have an IP address and a gateway, the application will not be able to function regularly. However, when that step is complete, normal service can resume. Any pages that were loading when the TCP/IP server went down will not finish loading. However, this can be "fixed" simply by reloading the pages.

B.4 Emulating `makefs`

The `makefs` command takes a directory tree and creates a file system image out of it. This groundbreaking utility was developed back when crossbuild capability was added to the NetBSD source tree. Since `makefs` constructs the file system

purely in userspace, it does not depend on the buildhost kernel to have file system support or the build process to have privileges to mount a file system. However, its implementation requires many one-way modifications to the kernel file system driver. Since performing these modifications is complicated, out of the NetBSD kernel file systems with r/w support makefs supports only FFS.

This part of the tutorial will show how to accomplish the same with out-of-the-box binaries. It applies to any r/w kernel file system for which NetBSD ships a **newfs**-type utility capable of creating image files. We learn how to mount a file system within the rump fs namespace and how to copy files to the file system image.

First, we need a suitable victim directory tree we want to create an image out of. We will again use the nethack source tree as an example. We need to find out how much space the directory tree will require.

```
golem> du -sh nethack-3.4.3/
12M     nethack-3.4.3/
```

Next, we need to create an empty file system. We use the standard **newfs** tool for this (command name will vary depending on target file system type). Since the file system must also accommodate metadata such as inodes and directory entries, we will create a slightly larger file system than what was indicated by **du** and reserve roughly 10% more disk space. There are ways to increase the accuracy of this calculation, but they are beyond the scope of this document.

```
golem> newfs -F -s 14M nethack.img
nethack.img: 14.0MB (28672 sectors) block size 4096, fragment size 512
        using 4 cylinder groups of 3.50MB, 896 blks, 1696 inodes.
super-block backups (for fsck_ffs -b #) at:
32, 7200, 14368, 21536,
```

Now, we need to start a rump server capable of mounting this particular file system type. As in the cgd example, we map the host image as `/dk` in the rump kernel namespace.

```
golem> rump_server -lrumpvfs -lrumpfs_ffs
    -d key=/dk,hostpath=nethack.img,size=host unix:///tmp/ffs_server
```

Next, we need to configure our shell for rump syscall hijacking. This is done by pointing the `LD_PRELOAD` environment variable to the hijack library. Every command executed with the variable set will attempt to contact the rump server and will fail if the server cannot contacted. This is demonstrated below by first omitting `RUMP_SERVER` and attempting to run a command. Shell builtins such as `export` and `unset` can still be run, since they do not start a new process.

```
golem> export LD_PRELOAD=/usr/lib/librumphijack.so
golem> lua -e 'print("Hello, rump!")'
lua: rumpclient init: No such file or directory
golem> export RUMP_SERVER=unix:///tmp/ffs_server
golem> lua -e 'print("Hello, rump!")'
Hello, rump!
```

Now, we can access the rump kernel file system namespace using the special path prefix `/rump`.

```
golem> ls -l /rump
total 1
drwxr-xr-x  2 root  wheel  512 Mar 12 13:31 dev
```

By default, a rump root file system includes only some autogenerated device nodes based on which components are loaded. As an experiment, you can try the above also against a server which does not support VFS.

We then proceed to create a mountpoint and mount the file system. Note, we start a new shell here because the one where we set `LD_PRELOAD` in was not *executed* with the variable set. That process does not have hijacking configured and we cannot `cd` into `/rump`. There is no reason we could not perform everything without changing the current working directory, but doing so often means less typing.

```
golem> $SHELL
golem> cd /rump
golem> mkdir mnt
golem> df -i mnt
Filesystem   1K-blocks       Used      Avail %Cap    iUsed   iAvail %iCap Mounted on
rumpfs               1          1          0 100%        0        0    0% /
golem> mount_ffs /dk /rump/mnt
mount_ffs: Warning: realpath /dk: No such file or directory
golem> df -i mnt
Filesystem   1K-blocks       Used      Avail %Cap    iUsed   iAvail %iCap Mounted on
/dk              13423          0      12752   0%        1     6781    0% /mnt
```

Note that the realpath warning from `mount_ffs` is only a warning and can be ignored. It is a result of the userland utility trying to find the source device `/dk`, but cannot since it is available only inside the rump kernel. Note that you need to supply the full path for the mountpoint, i.e. `/rump/mnt` instead of `mnt`. Otherwise the userland mount utility may adjust it incorrectly.

If you run the `mount` command you will note that the mounted file system is *not* present. This is expected, since the file system has been mounted within the rump kernel and not the host kernel, and therefore the host kernel does not know anything about it. The list of mounted file system is fetched with the `getvfsstat()` system call. Since the system call does not take any pathname, the hijacking library cannot automatically determine if the user wanted the mountpoints from the host kernel or the rump kernel. However, it is possible for the user to configure

the behavior by setting the RUMPHIJACK environment variable to contain the string vfs=getvfsstat.

```
golem> env RUMPHIJACK=vfs=getvfsstat mount
rumpfs on / type rumpfs (local)
/dk on /mnt type ffs (local)
```

Other ways of configuring the behavior of system call hijacking are described on the manual page. Note that setting the variable will override the default behavior, including the ability to access /rump. You can restore this by setting the variable to vfs=getvfsstat,path=/rump. Like with LD_PRELOAD, setting the variable will affect only processes you run after setting it, and the behavior of the shell it was set in will remain unchanged.

Now we can copy the files over. Due to how pax works, we first change our working directory to avoid encoding the full source path in the destination. The alternative is use us the -s option, but I find that changing the directory is often simpler.

```
golem> cd ~/srcdir
golem> pax -rw nethack-3.4.3 /rump/mnt/
golem> df -i /rump/mnt/
Filesystem   1K-blocks       Used      Avail %Cap    iUsed   iAvail %iCap Mounted on
/dk              13423      11962        790  93%      695     6087   10% /mnt
```

For comparison, we present the same operation using cp. Obviously, only one of pax or cp is necessary and you can use whichever you find more convenient.

```
golem> cp -Rp ~/srcdir/nethack-3.4.3 mnt/
golem> df -i /rump/mnt/
Filesystem   1K-blocks       Used      Avail %Cap    iUsed   iAvail %iCap Mounted on
/dk              13423      11962        790  93%      695     6087   10% /mnt
```

Then, the only thing left is to unmount the file system to make sure that we have a clean file system image.

```
golem> umount -R /rump/mnt
golem> df -i /rump/mnt
Filesystem   1K-blocks       Used      Avail %Cap    iUsed   iAvail %iCap Mounted on
rumpfs               1          1          0 100%        0        0    0% /
```

It is necessary to give the `-R` option to umount, or it will attempt to adjust the path by itself. This will usually result in the wrong path and the unmount operation failing. It is possible to set `RUMPHIJACK` in a way which does not require using `-R`, but that is left as an exercise for the reader.

We do not need to remove the mountpoint since the rump root file system is an in-memory file system and will be removed automatically when we halt the server.

Congratulations, you now have a clean file system image containing the desired files.

B.5 Master class: NFS server

This section presents scripts which allow to start a rump kernel capable of serving NFS file systems and how to mount the service using a client connected to another kernel server.

B.5.1 NFS Server

```
#!/bin/sh
#
```

```
# This script starts a rump kernel with NFS serving capability,
# configures a network interface and starts hijacked binaries
# which are necessary to serve NFS (rpcbind, mountd, nfsd).
#

# directory used for all temporary stuff
NFSX=/tmp/nfsx

# no need to edit below this line

haltserv()
{
        RUMP_SERVER=unix://${NFSX}/nfsserv rump.halt 2> /dev/null
        RUMP_SERVER=unix://${NFSX}/nfscli rump.halt 2> /dev/null
}

die()
{
        haltserv
        echo $*
        exit 1
}

# start from a fresh table
haltserv
rm -rf ${NFSX}
mkdir ${NFSX} || die cannot mkdir ${NFSX}

# create ffs file system we'll be exporting
newfs -F -s 10000 ${NFSX}/ffs.img > /dev/null || die could not create ffs

# start nfs kernel server.  this is a mouthful
export RUMP_SERVER=unix://${NFSX}/nfsserv
rump_server -lrumpvfs -lrumpdev -lrumpnet                               \
    -lrumpnet_net -lrumpnet_netinet -lrumpnet_local -lrumpnet_shmif     \
    -lrumpdev_disk -lrumpfs_ffs -lrumpfs_nfs -lrumpfs_nfsserver         \
```

```
    -d key=/dk,hostpath=${NFSX}/ffs.img,size=host ${RUMP_SERVER}
[ $? -eq 0 ] || die rump server startup failed

# configure server networking
rump.ifconfig shmif0 create
rump.ifconfig shmif0 linkstr ${NFSX}/shmbus
rump.ifconfig shmif0 inet 10.1.1.1

# especially rpcbind has a nasty habit of looping
export RUMPHIJACK_RETRYCONNECT=die
export LD_PRELOAD=/usr/lib/librumphijack.so

# "mtree"
mkdir -p /rump/var/run
mkdir -p /rump/var/db
touch /rump/var/db/mountdtab
mkdir /rump/etc
mkdir /rump/export

# create /etc/exports
echo '/export -noresvport -noresvmnt -maproot=0:0 10.1.1.100' |        \
    dd of=/rump/etc/exports 2> /dev/null

# mount our file system
mount_ffs /dk /rump/export 2> /dev/null || die mount failed
touch /rump/export/its_alive

# start rpcbind.  we want /var/run/rpcbind.sock
RUMPHIJACK='blanket=/var/run,socket=all' rpcbind || die rpcbind start

# ok, then we want mountd in the similar fashion
RUMPHIJACK='blanket=/var/run:/var/db:/export,socket=all,path=/rump,vfs=all' \
    mountd /rump/etc/exports || die mountd start

# finally, it's time for the infamous nfsd to hit the stage
RUMPHIJACK='blanket=/var/run,socket=all,vfs=all' nfsd -tu
```

B.5.2 NFS Client

```
#!/bin/sh
#
# This script starts a rump kernel which contains the drivers necessary
# to mount an NFS export.  It then proceeds to mount and provides
# a directory listing of the mountpoint.
#

NFSX=/tmp/nfsx

export RUMP_SERVER=unix://${NFSX}/nfscli
rump.halt 2> /dev/null
rump_server -lrumpvfs -lrumpnet -lrumpnet_net -lrumpnet_netinet        \
    -lrumpnet_shmif -lrumpfs_nfs ${RUMP_SERVER}

rump.ifconfig shmif0 create
rump.ifconfig shmif0 linkstr ${NFSX}/shmbus
rump.ifconfig shmif0 inet 10.1.1.100

export LD_PRELOAD=/usr/lib/librumphijack.so

mkdir /rump/mnt

mount_nfs 10.1.1.1:/export /rump/mnt

echo export RUMP_SERVER=unix://${NFSX}/nfscli
echo export LD_PRELOAD=/usr/lib/librumphijack.so
```

B.5.3 Using it

To use the NFS server, just run both scripts. The client script will print configuration data, so you can eval the script's output in a Bourne type shell for the correct

configuration.

```
golem> sh rumpnfsd.sh
golem> eval ‘sh rumpnfsclient.sh‘
```

That's it. You can start a shell and access the NFS client as normal.

```
golem> df /rump/mnt
Filesystem       1K-blocks       Used      Avail %Cap Mounted on
10.1.1.1:/export       4631          0       4399   0% /mnt
golem> sh
golem> cd /rump
golem> jot 100000 > mnt/numbers
golem> df mnt
Filesystem       1K-blocks       Used      Avail %Cap Mounted on
10.1.1.1:/export       4631        580       3819  13% /mnt
```

When you're done, stop the servers in the normal fashion. You may also want to remove the `/tmp/nfsx` temporary directory.

B.6 Further ideas

Kernel code development and debugging was a huge personal motivation for working on this, and is a truly excellent use case especially if you want to safely and easily learn about how various parts of the kernel work.

There are also more user-oriented applications. For example, you can construct servers which run hardware drivers from some later release of NetBSD than what

is running on your host. You can also distribute these devices as services on the network.

On a multiuser machine where you do not have control over how your data is backed up you can use a cgd server to provide a file system with better confidentiality guarantees than your regular home directory. You can easily configure your applications to communicate directly with the cryptographic server, and confidential data will never hit the disk unencrypted. This, of course, does not protect against all threat models on a multiuser system, but is a simple way of protecting yourself against one of them.

Furthermore, you have more fine grained control over privileges. For example, opening a raw socket requires root privileges. This is still true for a rump server, but the difference is that it requires root privileges in the rump kernel, not the host kernel. Now, if rump server runs with normal user privileges (as is recommended), you cannot use rump kernel root privileges for full control of the hosting OS.

In the end, this document only scratched the surface of what is possible by running kernel code as services in userspace.

Appendix C Patches to the 5.99.48 source tree

Fix a compilation error in code which is not compiled by default:

```
Index: sys/rump/librump/rumpkern/locks_up.c
===================================================================
RCS file: /usr/allsrc/repo/src/sys/rump/librump/rumpkern/locks_up.c,v
retrieving revision 1.5
diff -p -u -r1.5 locks_up.c
--- sys/rump/librump/rumpkern/locks_up.c        1 Dec 2010 17:22:51 -0000       1.5
+++ sys/rump/librump/rumpkern/locks_up.c        2 Dec 2011 16:40:00 -0000
@@ -160,6 +160,7 @@ mutex_owned(kmutex_t *mtx)
 struct lwp *
 mutex_owner(kmutex_t *mtx)
 {
+       UPMTX(mtx);

        return upm->upm_owner;
 }
```

Fix a locking error in a branch which is not hit during normal execution:

```
Index: sys/rump/librump/rumpkern/vm.c
===================================================================
RCS file: /usr/allsrc/repo/src/sys/rump/librump/rumpkern/vm.c,v
retrieving revision 1.114
diff -u -r1.114 vm.c
--- sys/rump/librump/rumpkern/vm.c        21 Mar 2011 16:41:08 -0000      1.114
+++ sys/rump/librump/rumpkern/vm.c        12 Dec 2011 14:16:02 -0000
@@ -1132,7 +1132,6 @@
                        rumpuser_dprintf("pagedaemoness: failed to reclaim "
                            "memory ... sleeping (deadlock?)\n");
                        cv_timedwait(&pdaemoncv, &pdaemonmtx, hz);
-                       mutex_enter(&pdaemonmtx);
                }
        }
```

This is a quick fix for making it possible to link rump kernels which do not include the VFS faction, but include components which want to create device nodes. A more thorough fix should not use weak symbols and examine the call sites as well — the stubs the calls are aliased to return a failure, and not all callers tolerate that.

```
Index: sys/rump/librump/rumpkern/rump.c
===================================================================
RCS file: /usr/allsrc/repo/src/sys/rump/librump/rumpkern/rump.c,v
retrieving revision 1.234
diff -p -u -r1.234 rump.c
--- sys/rump/librump/rumpkern/rump.c      22 Mar 2011 15:16:23 -0000      1.234
+++ sys/rump/librump/rumpkern/rump.c      5 Jan 2012 23:42:17 -0000
@@ -160,6 +161,9 @@
 rump_proc_vfs_init_fn rump_proc_vfs_init;
 rump_proc_vfs_release_fn rump_proc_vfs_release;

+__weak_alias(rump_vfs_makeonedevnode,rump__unavailable);
+__weak_alias(rump_vfs_makedevnodes,rump__unavailable);
+
 static void add_linkedin_modules(const struct modinfo *const *, size_t);

 static void __noinline
```

The following patch fixes the faster I/O mode for ukfs(3). It was written to produce the minimal diff.

```
Index: lib/libukfs/ukfs.c
===================================================================
RCS file: /cvsroot/src/lib/libukfs/ukfs.c,v
retrieving revision 1.57
diff -p -u -r1.57 ukfs.c
--- lib/libukfs/ukfs.c  22 Feb 2011 15:42:15 -0000      1.57
+++ lib/libukfs/ukfs.c  5 Jul 2012 20:53:34 -0000
@@ -115,14 +115,18 @@ ukfs_getspecific(struct ukfs *ukfs)
 #endif

 static int
-precall(struct ukfs *ukfs, struct lwp **curlwp)
+precall(struct ukfs *ukfs, struct lwp **curlwp, bool sharefd)
```

```
 {
+       int rfflags = 0;
+
+       if (!sharefd)
+               rfflags = RUMP_RFCFDG;

        /* save previous.  ensure start from pristine context */
        *curlwp = rump_pub_lwproc_curlwp();
        if (*curlwp)
                rump_pub_lwproc_switch(ukfs->ukfs_lwp);
-       rump_pub_lwproc_rfork(RUMP_RFCFDG);
+       rump_pub_lwproc_rfork(rfflags);

        if (rump_sys_chroot(ukfs->ukfs_mountpath) == -1)
                return errno;
@@ -145,7 +149,17 @@ postcall(struct lwp *curlwp)
 struct lwp *ukfs_curlwp;                                                \
 do {                                                                    \
        int ukfs_rv;                                                    \
-       if ((ukfs_rv = precall(ukfs, &ukfs_curlwp)) != 0) {             \
+       if ((ukfs_rv = precall(ukfs, &ukfs_curlwp, false)) != 0) {      \
+               errno = ukfs_rv;                                        \
+               return -1;                                              \
+       }                                                               \
+} while (/*CONSTCOND*/0)
+
+#define PRECALL2()                                                      \
+struct lwp *ukfs_curlwp;                                                \
+do {                                                                    \
+       int ukfs_rv;                                                    \
+       if ((ukfs_rv = precall(ukfs, &ukfs_curlwp, true)) != 0) {       \
                errno = ukfs_rv;                                        \
                return -1;                                              \
        }                                                               \
@@ -848,7 +862,7 @@ ukfs_open(struct ukfs *ukfs, const char
 {
        int fd;

-       PRECALL();
+       PRECALL2();
        fd = rump_sys_open(filename, flags, 0);
        POSTCALL();
        if (fd == -1)
```

This fixes the conditionally compiled block device layer host direct I/O support.

```
Index: sys/rump/librump/rumpvfs/rumpblk.c
===================================================================
RCS file: /usr/allsrc/repo/src/sys/rump/librump/rumpvfs/rumpblk.c,v
retrieving revision 1.46
diff -p -u -r1.46 rumpblk.c
--- sys/rump/librump/rumpvfs/rumpblk.c  3 Feb 2011 22:16:11 -0000       1.46
+++ sys/rump/librump/rumpvfs/rumpblk.c  5 Jul 2012 21:05:18 -0000
@@ -109,9 +109,7 @@ static struct rblkdev {
        char *rblk_path;
        int rblk_fd;
        int rblk_mode;
-#ifdef HAS_ODIRECT
        int rblk_dfd;
-#endif
        uint64_t rblk_size;
        uint64_t rblk_hostoffset;
        uint64_t rblk_hostsize;
@@ -368,7 +366,7 @@ rumpblk_init(void)
        for (i = 0; i < RUMPBLK_SIZE; i++) {
                mutex_init(&minors[i].rblk_memmtx, MUTEX_DEFAULT, IPL_NONE);
                cv_init(&minors[i].rblk_memcv, "rblkmcv");
-               minors[i].rblk_fd = -1;
+               minors[i].rblk_fd = minors[i].rblk_dfd = -1;
        }

        evcnt_attach_dynamic(&ev_io_total, EVCNT_TYPE_MISC, NULL,
@@ -501,6 +499,9 @@ static int
 backend_open(struct rblkdev *rblk, const char *path)
 {
        int error, fd;
+#ifdef HAS_ODIRECT
+       int dummy;
+#endif

        KASSERT(rblk->rblk_fd == -1);
        fd = rumpuser_open(path, O_RDWR, &error);
@@ -514,7 +515,7 @@ backend_open(struct rblkdev *rblk, const
                rblk->rblk_dfd = rumpuser_open(path,
                    O_RDONLY | O_DIRECT, &error);
                if (error) {
-                       close(fd);
+                       rumpuser_close(fd, &dummy);
```

```
                        return error;
                }
 #endif
@@ -525,13 +526,13 @@ backend_open(struct rblkdev *rblk, const
                rblk->rblk_dfd = rumpuser_open(path,
                    O_RDWR | O_DIRECT, &error);
                if (error) {
-                       close(fd);
+                       rumpuser_close(fd, &dummy);
                        return error;
                }
 #endif
        }

-       if (rblk->rblk_ftype == RUMPUSER_FT_REG) {
+       if (rblk->rblk_ftype == RUMPUSER_FT_REG && rblk->rblk_dfd != -1) {
                uint64_t fsize= rblk->rblk_hostsize, off= rblk->rblk_hostoffset;
                struct blkwin *win;
                int i, winsize;
@@ -591,12 +592,10 @@ backend_close(struct rblkdev *rblk)
        rumpuser_fsync(rblk->rblk_fd, &dummy);
        rumpuser_close(rblk->rblk_fd, &dummy);
        rblk->rblk_fd = -1;
-#ifdef HAS_ODIRECT
        if (rblk->rblk_dfd != -1) {
                rumpuser_close(rblk->rblk_dfd, &dummy);
                rblk->rblk_dfd = -1;
        }
-#endif

        return 0;
 }
```

Aalto-DD 171/2012

ISBN 978-952-60-4916-8
ISBN 978-952-60-4917-5 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science and Engineering
www.aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL DISSERTATIONS