Xen

PRENTICE HALL

PRENTICE HALL OPEN SOURCE SOFTWARE DEVELOPMENT SERIES

The Definitive Guide to the Xen Hypervisor

"The Xen community is leading the industry forward in virtualization, and this book will play an important role in helping it to grow and develop both the Xen hypervisor and products that deliver it to market."

—From the foreword by Ian Pratt

David Chisnall

Foreword by Ian Pratt, Xen Project Lead and Founder of XenSource

The Definitive Guide to the Xen Hypervisor

Prentice Hall
Open Source Software Development Series

Arnold Robbins, Series Editor

"Real world code from real world applications"

Open Source technology has revolutionized the computing world. Many large-scale projects are in production use worldwide, such as Apache, MySQL, and Postgres, with programmers writing applications in a variety of languages including Perl, Python, and PHP. These technologies are in use on many different systems, ranging from proprietary systems, to Linux systems, to traditional UNIX systems, to mainframes.

The **Prentice Hall Open Source Software Development Series** is designed to bring you the best of these Open Source technologies. Not only will you learn how to use them for your projects, but you will learn *from* them. By seeing real code from real applications, you will learn the best practices of Open Source developers the world over.

Titles currently in the series include:

Linux® Debugging and Performance Tuning
Steve Best
0131492470, Paper, ©2006

The Definitive Guide to the Xen Hypervisor
David Chisnall
013234971X, Hard, ©2008

Understanding AJAX
Joshua Eichorn
0132216353, Paper, ©2007

The Linux Programmer's Toolbox
John Fusco
0132198576, Paper, ©2007

Embedded Linux Primer
Christopher Hallinan
0131679848, Paper, ©2007

The Apache Modules Book
Nick Kew
0132409674, Paper, © 2007

SELinux by Example
Frank Mayer, David Caplan, Karl MacMillan
0131963694, Paper, ©2007

UNIX to Linux® Porting
Alfredo Mendoza, Chakarat Skawratananond, Artis Walker
0131871099, Paper, ©2006

Rapid Web Applications with TurboGears
Mark Ramm, Kevin Dangoor, Gigi Sayfan
0132433885, Paper, © 2007

Linux Programming by Example
Arnold Robbins
0131429647, Paper, ©2004

The Linux® Kernel Primer
Claudia Salzberg, Gordon Fischer, Steven Smolski
0131181637, Paper, ©2006

Rapid GUI Programming with Python and Qt
Mark Summerfield
0132354187, Hard, © 2008

New to the series: Digital Short Cuts

Short Cuts are short, concise, PDF documents designed specifically for busy technical professionals like you. Each Short Cut is tightly focused on a specific technology or technical problem. Written by industry experts and best selling authors, Short Cuts are published with you in mind — getting you the technical information that you need — now.

Understanding AJAX:
Consuming the Sent Data with XML and JSON
Joshua Eichorn
0132337932, Adobe Acrobat PDF, © 2007

Debugging Embedded Linux
Christopher Hallinan
0131580132, Adobe Acrobat PDF, © 2007

Using BusyBox
Christopher Hallinan
0132335921, Adobe Acrobat PDF, © 2007

The Definitive Guide to the Xen Hypervisor

David Chisnall

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Visit us on the Web: www.prenhallprofessional.com

Library of Congress Cataloging-in-Publication Data
Chisnall, David.
The definitive guide to the Xen hypervisor / David Chisnall.
p. cm.
Includes index.
ISBN-13: 978-0-13-234971-0 (hardcover : alk. paper) 1. Xen (Electronic resource) 2. Virtual computer systems. 3. Computer organization. 4. Parallel processing (Electronic computers) I. Title.
QA76.9.V5C427 2007
005.4'3—dc22
2007036152

ISBN-13: 978-0-13-234971-0
ISBN-10: 0-13-234971-X

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, November 2007

Editor-in-Chief
Mark L. Taub

Acquisitions Editor
Debra Williams Cauley

Development Editor
Michael Thurston

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Technical Reviewer
Glenn Tremblay

Cover Designer
Alan Clements

Composition
David Chisnall

Contents

List of Figures xi

List of Tables xiii

Foreword xv

Preface xvii

I The Xen Virtual Machine 1

1 The State of Virtualization 3
- 1.1 What Is Virtualization? . . . 3
  - 1.1.1 CPU Virtualization . . . 4
  - 1.1.2 I/O Virtualization . . . 5
- 1.2 Why Virtualize? . . . 7
- 1.3 The First Virtual Machine . . . 8
- 1.4 The Problem of x86 . . . 9
- 1.5 Some Solutions . . . 9
  - 1.5.1 Binary Rewriting . . . 10
  - 1.5.2 Paravirtualization . . . 10
  - 1.5.3 Hardware-Assisted Virtualization . . . 13
- 1.6 The Xen Philosophy . . . 15
  - 1.6.1 Separation of Policy and Mechanism . . . 15
  - 1.6.2 Less Is More . . . 15
- 1.7 The Xen Architecture . . . 16
  - 1.7.1 The Hypervisor, the OS, and the Applications . . . 16
  - 1.7.2 The Rôle of Domain 0 . . . 19
  - 1.7.3 Unprivileged Domains . . . 22
  - 1.7.4 HVM Domains . . . 22
  - 1.7.5 Xen Configurations . . . 23

2 Exploring the Xen Virtual Architecture **27**
2.1 Booting as a Paravirtualized Guest 27
2.2 Restricting Operations with Privilege Rings 28
2.3 Replacing Privileged Instructions with Hypercalls 30
2.4 Exploring the Xen Event Model 33
2.5 Communicating with Shared Memory 34
2.6 Split Device Driver Model 35
2.7 The VM Lifecycle 37
2.8 Exercise: The Simplest Xen Kernel 38
2.8.1 The Guest Entry Point 40
2.8.2 Putting It All Together 43

3 Understanding Shared Info Pages **47**
3.1 Retrieving Boot Time Info 47
3.2 The Shared Info Page 51
3.3 Time Keeping in Xen 53
3.4 Exercise: Implementing gettimeofday() 54

4 Using Grant Tables **59**
4.1 Sharing Memory 59
4.1.1 Mapping a Page Frame 61
4.1.2 Transferring Data between Domains 63
4.2 Device I/O Rings 65
4.3 Granting and Revoking Permissions 66
4.4 Exercise: Mapping a Granted Page 69
4.5 Exercise: Sharing Memory between VMs 71

5 Understanding Xen Memory Management **75**
5.1 Managing Memory with x86 75
5.2 Pseudo-Physical Memory Model 78
5.3 Segmenting on 32-bit x86 80
5.4 Using Xen Memory Assists 82
5.5 Controlling Memory Usage with the Balloon Driver 84
5.6 Other Memory Operations 86
5.7 Updating the Page Tables 89
5.7.1 Creating a New VM Instance 93
5.7.2 Handling a Page Fault 94
5.7.3 Suspend, Resume, and Migration 94
5.8 Exercise: Mapping the Shared Info Page 95

II Device I/O 97

6 Understanding Device Drivers 99
- 6.1 The Split Driver Model . . . 100
- 6.2 Moving Drivers out of Domain 0 . . . 102
- 6.3 Understanding Shared Memory Ring Buffers . . . 103
  - 6.3.1 Examining the Xen Implementation . . . 105
  - 6.3.2 Ordering Operations with Memory Barriers . . . 107
- 6.4 Connecting Devices with XenBus . . . 109
- 6.5 Handling Notifications from Events . . . 111
- 6.6 Configuring via the XenStore . . . 112
- 6.7 Exercise: The Console Device . . . 112

7 Using Event Channels 119
- 7.1 Events and Interrupts . . . 119
- 7.2 Handling Traps . . . 120
- 7.3 Event Types . . . 123
- 7.4 Requesting Events . . . 124
- 7.5 Binding an Event Channel to a VCPU . . . 127
- 7.6 Operations on Bound Channels . . . 128
- 7.7 Getting a Channel's Status . . . 129
- 7.8 Masking Events . . . 130
- 7.9 Events and Scheduling . . . 132
- 7.10 Exercise: A Full Console Driver . . . 133

8 Looking through the XenStore 141
- 8.1 The XenStore Interface . . . 141
- 8.2 Navigating the XenStore . . . 142
- 8.3 The XenStore Device . . . 145
- 8.4 Reading and Writing a Key . . . 147
  - 8.4.1 The Userspace Way . . . 148
  - 8.4.2 From the Kernel . . . 150
- 8.5 Other Operations . . . 158

9 Supporting the Core Devices 161
- 9.1 The Virtual Block Device Driver . . . 161
  - 9.1.1 Setting Up the Block Device . . . 162
  - 9.1.2 Data Transfer . . . 165
- 9.2 Using Xen Networking . . . 169
  - 9.2.1 The Virtual Network Interface Driver . . . 169
  - 9.2.2 Setting Up the Virtual Interface . . . 169
  - 9.2.3 Sending and Receiving . . . 170

9.2.4 NetChannel2 174

10 Other Xen Devices **177**
10.1 CD Support 177
10.2 Virtual Frame Buffer 178
10.3 The TPM Driver 183
10.4 Native Hardware 184
10.4.1 PCI Support 184
10.4.2 USB Devices 186
10.5 Adding a New Device Type 187
10.5.1 Advertising the Device 187
10.5.2 Setting Up Ring Buffers 188
10.5.3 Difficulties 189
10.5.4 Accessing the Device 191
10.5.5 Designing the Back End 191

III Xen Internals **195**

11 The Xen API **197**
11.1 XML-RPC 198
11.1.1 XML-RPC Data Types 198
11.1.2 Remote Procedure Calls 199
11.2 Exploring the Xen Interface Hierarchy 200
11.3 The Xen API Classes 201
11.3.1 The C Bindings 203
11.4 The Function of Xend 206
11.5 Xm Command Line 208
11.6 Xen CIM Providers 209
11.7 Exercise: Enumerating Running VMs 210
11.8 Summary 215

12 Virtual Machine Scheduling **217**
12.1 Overview of the Scheduler Interface 218
12.2 Historical Schedulers 219
12.2.1 SEDF 221
12.2.2 Credit Scheduler 222
12.3 Using the Scheduler API 224
12.3.1 Running a Scheduler 225
12.3.2 Domain 0 Interaction 228
12.4 Exercise: Adding a New Scheduler 229
12.5 Summary 233

13 HVM Support **235**
13.1 Running Unmodified Operating Systems . . . 235
13.2 Intel VT-x and AMD SVM . . . 237
13.3 HVM Device Support . . . 239
13.4 Hybrid Virtualization . . . 240
13.5 Emulated BIOS . . . 244
13.6 Device Models and Legacy I/O Emulation . . . 245
13.7 Paravirtualized I/O . . . 246
13.8 HVM Support in Xen . . . 248

14 Future Directions **253**
14.1 Real to Virtual, and Back Again . . . 253
14.2 Emulation and Virtualization . . . 254
14.3 Porting Efforts . . . 255
14.4 The Desktop . . . 257
14.5 Power Management . . . 259
14.6 The Domain 0 Question . . . 261
14.7 Stub Domains . . . 263
14.8 New Devices . . . 264
14.9 Unusual Architectures . . . 265
14.10 The Big Picture . . . 267

IV Appendix **271**

PV Guest Porting Cheat Sheet **273**
A.1 Domain Builder . . . 273
A.2 Boot Environment . . . 274
A.3 Setting Up the Virtual IDT . . . 274
A.4 Page Table Management . . . 275
A.5 Drivers . . . 276
A.6 Domain 0 Responsibilities . . . 276
A.7 Efficiency . . . 277
A.8 Summary . . . 278

Index **279**

This page intentionally left blank

List of Figures

1.1 An instruction stream in a VM 11
1.2 System calls in native and paravirtualized systems 12
1.3 Ring usage in native and paravirtualized systems 17
1.4 Ring usage in x86-64 native and paravirtualized systems 18
1.5 The path of a packet sent from an unprivileged guest through the system 20
1.6 A simple Xen configuration 24
1.7 A Xen configuration showing driver isolation and an unmodified guest OS 25
1.8 A single node in a clustered Xen environment 25

2.1 The lifecycle of a real machine 37
2.2 The lifecycle of a virtual machine 38

3.1 The hierarchy of structures used for the shared info page 51

4.1 The structure of an I/O ring 67

5.1 The three layers of Xen memory 80
5.2 Memory layout on x86 systems 81

6.1 The composition of a split device driver 101
6.2 A sequence of actions on a ring buffer 104

7.1 The process of delivering an event 131

11.1 The Xen interface hierarchy 201
11.2 Objects associated with a host 202
11.3 Objects associated with a VM instance 203

This page intentionally left blank

List of Tables

2.1 Xen components and their UNIX counterparts 34

4.1 Grant table status codes 63

5.1 Segment descriptors on x86 76
5.2 Available VM assists 84
5.3 Extended MMU operation commands 92

7.1 Event channel status values 130

This page intentionally left blank

Foreword

With the recent release of Xen 3.1 the Xen community has delivered the world's most advanced hypervisor, which serves as an open source industry standard for virtualization. The Xen community benefits from the support of over 20 of the world's leading IT vendors, contributions from vendors and research groups worldwide, and is the driving force of innovation in virtualization in the industry.

The continued growth and excellence of Xen is a vindication of the project's component strategy. Rather than developing a complete open source product, the project endorses an integrated approach whereby the Xen hypervisor is included as the virtualization "engine" in multiple products and projects. For example, Xen is delivered as an integrated hypervisor with many operating systems, including Linux, Solaris, and BSD, and is also packaged as virtualizaton platforms such as XenSource's XenEnterprise. This allows Xen to serve many different use cases and customer needs for virtualization.

Xen supports a wide range of architectures, from super-computer systems with thousands of Intel Itanium CPUs, to Power PC and industry standard x86 servers and clients, and even ARM-9 based PDAs. The project's cross-architecture, multi-OS approach to virtualization is another of its key strengths, and has enabled it to influence the design of proprietary products, including the forthcoming Microsoft Windows Hypervisor, and benefit from hardware-assisted virtualization technologies from CPU, chipset, and fabric vendors. The project also works actively in the DMTF, to develop industry standard management frameworks for virtualized systems.

The continued success of the Xen hypervisor depends substantially on the development of a highly skilled community of developers who can both contribute to the project and use the technology within their own products. To date, other than the community's limited documentation, and a steep learning curve for the uninitiated, Xen has retained a mystique that is unmistakably "cool" but not scalable. While there are books explaining how to use Xen in the context of particular vendors' products, there is a huge need for a definitive technical insider's guide to the Xen hypervisor itself. Continuing the "engine" analogy, there are books available for "cars" that integrate Xen, but no manuals on how to fix the

"engine." The publication of this book is therefore of great importance to the Xen community and the industry of vendors around it.

David Chisnall brings to this project the deep systems expertise that is required to dive deep inside Xen, understand its complex subsystems, and document its workings. With a Ph.D. in computer science, and as an active systems software developer, David has concisely distilled the complexity of Xen into a work that will allow a skilled systems developer to get a firm grip on how Xen works, how it interfaces to key hardware systems, and even how to develop it. To complete his work, David spent a considerable period of time with the XenSource core team in Cambridge, U.K., where he developed a unique insight into the history, architecture, and inner workings of Xen. Without doubt his is the most thorough in-depth book on the Xen hypervisor available, and fully merits its description as the definitive insider's guide.

It is my hope and belief that this work will contribute significantly to the continued development of the Xen project, and the adoption of Xen worldwide. The opportunity for open source virtualization is huge, and the open source community is the foundation upon which rapid innovation and delivery of differentiated solutions is founded. The Xen community is leading the industry forward in virtualization, and this book will play an important role in helping it to grow and develop both the Xen hypervisor and products that deliver it to market.

Ian Pratt
Xen Project Lead and Founder of XenSource

Preface

This book aims to serve as a guide to the Xen hypervisor. The interface to paravirtualized guests is described in detail, along with some description of the internals of the hypervisor itself.

Any book about an open source project will, by nature, be less detailed than the code of the project that it attempts to describe. Anyone wishing to fully understand the Xen hypervisor will find no better source of authoritative information than the code itself. This book aims to provide a guided tour, indicating features of interest to help visitors find their way around the code. As with many travel books, it is to be hoped that readers will find it an informative read whether or not they visit the code.

Much of the focus of this book is on the kernel interfaces provided by Xen. Anyone wishing to write code that runs on the Xen hypervisor will find this material relevant, including userspace program developers wanting to take advantage of hypervisor-specific features.

Overview and Organization

This book is divided into three parts. The first two describe the hypervisor interfaces, while the last looks inside Xen itself.

Part I begins with a description of the history and current state of virtualization, including the conditions that caused Xen to be created, and an overview of the design decisions made by the developers of the hypervisor. The remainder of this part describes the core components of the virtual environment, which must be supported by any non-trivial guest kernel.

The second part focuses on device support for paravirtualized and paravirtualization-aware kernels. Xen provides an abstract interface to devices, built on some core communication systems provided by the hypervisor. Virtual equivalents of interrupts and DMA and the mechanism used for device discovery are all described in Part II, along with the interfaces used by specific device categories.

Part III takes a look at how the management tools interact with the hypervisor. It looks inside Xen to see how it handles scheduling of virtual machines, and how it uses CPU-specific features to support unmodified guests.

An appendix provides a quick reference for people wishing to port operating systems to run atop Xen.

Typographical Conventions

This book uses a number of different typefaces and other visual hints to describe different types of material.

Filenames, such as `/bin/sh`, are all shown in `this font`. This same convention is also used for structures which closely resemble a filesystem, such as paths in the XenStore.

Variable or function names, such as example(), used in text will be typeset like this. Registers, such as **EAX**, and instructions, such as **POP** will be shown in uppercase lettering. Single line listings will appear like this:

```
eg = example_function(arg1);
```

Longer listings will have line numbers down the left, and a gray background, as shown in Listing 1. In all listings, bold is used to indicate keywords, and italicized text represents strings and comments.

Listing 1: An example listing [from: example/hello.c]

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     /* Print hello world */
6     printf("Hello World!\n");
7     return 0;
8 }
```

Listings which are taken from external files will retain the line numbers of the original file, allowing the referenced section to be found easily by the reader. The captions contain the original source in square brackets. Those beginning with `example/` are from the example sources. All others, unless otherwise specified, are from the Xen sources.

Comments from files in the Xen source code have been preserved, complete with errors. Since the Xen source code predominantly uses U.K. English for comments, and variable and function names, this convention has been preserved in examples from this book.

During the course of this book, a simple example kernel is constructed. The source code for this can be downloaded from:

http://www.prenhallprofessional.com/title/9780132349710.

Output from command-line interaction is shown in the following way:

```
$ gcc hello.c
$ ./a.out
Hello World!
```

A `$` prompt indicates commands that can be run as any user, while a `#` is used to indicate that root access is likely to be required.

Use as a Text

In addition to the traditional uses for hypervisors, Xen makes an excellent teaching tool. Early versions of Xen only supported paravirtualized guests, and newer ones continue to support these in addition to unmodified guests. The architecture exposed by the hypervisor to paravirtualized guests is very similar to x86, but differs in a number of ways. Driver support is considerably easier, with a single abstract device being exposed for each device category, for example. In spite of this, a number of things are very similar. A guest operating system must handle interrupts (or their virtual equivalent), manage page tables, schedule running tasks, etc.

This makes Xen an excellent platform for development of new operating systems. Unlike a number of simple emulated systems, a guest running atop Xen can achieve performance within 10% that of the native host. The simple device interfaces make it easy for Xen guests to support devices, without having to worry about the multitude of peripherals available for real machines.

The similarity to real hardware makes Xen an ideal platform for teaching operating systems concepts. Writing a simple kernel that runs atop Xen is a significantly easier task than writing one that runs on real hardware, and significantly more rewarding than writing one that runs in a simplified machine emulator.

An operating systems course should use this text in addition to a text on general operating systems principles to provide the platform-specific knowledge required for students to implement their own kernels.

Xen is also a good example of a successful, modern, microkernel (although it does more in kernelspace than many microkernels), making it a good example for contrasting with popular monolithic systems.

Acknowledgments

First, I have to thank Mark Taub for the opportunity to write this book. Since first contacting Mark in 2002, he has given me the opportunity to work on several

projects. This included working with Mark Sobell, from whom I learned a lot about writing.

I also have to thank Debra Williams Cauley who coordinated everything for this book, along with the rest of her team who helped to transform it into the form you are now seeing.

I began writing this book near the end of the third year of my Ph.D., and would like to thank my supervisor, Professor Min Chen, for his forbearance when my thesis became a lower priority than getting this book finished. I would also like to thank the other members of the Swansea University Computer Science Department who kept me supplied with coffee while I was writing.

For technical assistance, I could have had no one more patient than Keir Fraser who answered my questions in great detail by email and in person when I visited XenSource. Without his help, this book would have taken a lot longer to write. A number of other people at XenSource and at the Spring 2007 XenSummit also provided valuable advice. I'd like to thank all of the people doing exciting things with Xen for helping to make this book so much fun to write.

I would also like to thank Glenn Tremblay of Marathon Technologies Corp. who performed a detailed technical review. While I can't guarantee that this book is error free, I can be very sure it wouldn't have been without his assistance. Glenn is a member of a growing group of people using Xen as a foundation for their own products, and I hope his colleagues find this book useful.

This book was written entirely in Vim. Subversion was used for revision tracking and the final manuscript was typeset using LaTeX. Without the work of Bram Moolenaar, Leslie Lamport, Donald Knuth, and many others, writing a book using Free Software would be much harder, if not impossible.

Finally, I would like to thank all of the members of the Slashdot community for helping me to procrastinate when I should have been writing.

Part I

The Xen Virtual Machine

This page intentionally left blank

Chapter 1

The State of Virtualization

Xen is a virtualization tool, but what does this mean? In this chapter, we will explore some of the history of virtualization, and some of the reasons why people found, and continue to find, it useful. We will have a look in particular at the x86, or IA32, architecture, why it presents such a problem for virtualization, and some possible ways around these limitations from other virtualization systems and finally from Xen itself.

1.1 What Is Virtualization?

Virtualization is very similar conceptually to emulation. With emulation, a system pretends to be another system. With virtualization, a system pretends to be two or more of the same system.

Most modern operating systems contain a simplified system of virtualization. Each running process is able to act as if it is the only thing running. The CPUs and memory are virtualized. If a process tries to consume all of the CPU, a modern operating system will preempt it and allow others their fair share. Similarly, a running process typically has its own virtual address space that the operating system maps to physical memory to give the process the illusion that it is the only user of RAM.

Hardware devices are also often virtualized by the operating system. A process can use the Berkeley Sockets API, or an equivalent, to access a network device without having to worry about other applications. A windowing system or virtual terminal system provides similar multiplexing to the screen and input devices.

Since you already use some form of virtualization every day, you can see that it is useful. The isolation it gives often prevents a bug, or intentionally malicious behavior, in one application from breaking others.

Unfortunately, applications are not the only things to contain bugs. Operating systems do too, and often these allow one application to compromise the isolation that it usually experiences. Even in the absence of bugs, it is often convenient to provide a greater degree of isolation than an operating system can.

1.1.1 CPU Virtualization

Virtualizing a CPU is, to some extent, very easy. A process runs with exclusive use of it for a while, and is then interrupted. The CPU state is then saved, and another process runs. After a while, this process is repeated.

This process typically occurs every 10ms or so in a modern operating system. It is worth noting, however, that the virtual CPU and the physical CPU are not identical. When the operating system is running, swapping processes, the CPU runs in a privileged mode. This allows certain operations, such as access to memory by physical address, that are not usually permitted. For a CPU to be completely virtualized, Popek and Goldberg put forward a set of requirements that must be met in their 1974 paper "Formal Requirements for Virtualizable Third Generation Architectures."[1] They began by dividing instructions into three categories:

Privileged instructions are defined as those that may execute in a privileged mode, but will trap if executed outside this mode.

Control sensitive instructions are those that attempt to change the configuration of resources in the system, such as updating virtual to physical memory mappings, communicating with devices, or manipulating global configuration registers.

Behavior sensitive instructions are those that behave in a different way depending on the configuration of resources, including all load and store operations that act on virtual memory.

In order for an architecture to be virtualizable, Popek and Goldberg determined that all sensitive instructions must also be privileged instructions. Intuitively, this means that a hypervisor must be able to intercept any instructions that change the state of the machine in a way that impacts other processes.

One of the easiest architectures to virtualize was the DEC[2] Alpha. The Alpha didn't have privileged instructions in the normal sense. It had one special instruction that jumped to a specified firmware ('PALCode') address and entered a special mode where some usually hidden registers were available.

[1] Published in *Communications of the ACM*.

[2] Digital Equipment Corporation (DEC) was later renamed Digital, then was bought by HP, which later merged with Compaq.

Once in this mode, the CPU could not be preempted. It would execute a sequence of normal instructions and then another instruction would return the CPU to the original mode. To perform context switches into the kernel, the userspace code would raise an exception, causing an automatic jump to PALCode. This would set a flag in a hidden register and then pass control to the kernel. The kernel could then call other PALCode instructions, which would check the value of the flag and permit special features to be accessed, before finally calling a PALCode instruction that would unset the flag and return control to the userspace program. This mechanism could be extended to provide the equivalent of multiple levels of privilege fairly easily by setting the privilege level in a hidden register, and checking it at the start of any PALCode routines.

Everything normally implemented as a privileged instruction was performed as a set of instructions stored in the PALCode. If you wanted to virtualize the Alpha, all you needed to do was replace the PALCode with a set of instructions that passed the operations through an abstraction layer.

1.1.2 I/O Virtualization

An operating system requires more than a CPU to run; it also depends on main memory, and a set of devices. Virtualizing memory is relatively easy; it can just be partitioned and every privileged instruction that accesses physical memory trapped and replaced with one that maps to the permitted range. A modern CPU includes a *Memory Management Unit* (*MMU*), which performs these translations, typically based on information provided by an operating system.

Other devices are somewhat more complicated. Most are not designed with virtualization in mind, and for some it is not entirely obvious how virtualization would be supported. A block device, such as a hard disk, could potentially be virtualized in the same way as main memory—by dividing it up into partitions that can be accessed by each virtual machine (VM). A graphics card, however, is a more complex problem. A simple frame buffer might be handled trivially by providing a virtual frame buffer to each VM and allow the user to either switch between them or map them into ranges on a physical display.

Modern graphics cards, however, are a lot more complicated than frame buffers; they provide 2D and 3D acceleration, and have a lot of internal state. Worse, most don't provide a mechanism for saving and restoring this state, and so even switching between VMs is problematic. This has already been a problem for people working on power management. If you are running a GUI, such as X11, some state may be stored in the graphics hardware—the current video mode, at the very least—which will be lost when the device is powered down. This means that the GUI must be modified to ensure that it also saves the state elsewhere, and can restore it when required (for example, by instructing every window to redraw

itself). This is obviously not possible for a true virtual environment, because the virtualized system is not aware that it has been disconnected from the hardware.

Another issue comes from the way in which devices interact with the system. Typically, data is transferred to and from devices via *Direct Memory Access* (*DMA*) transfers. The device is given a physical memory address by the driver and writes a chunk of data there. Because the device exists outside the normal framework of the operating system, it must use physical memory rather than a virtual address space.

This works fine if the operating system really is in complete control of the platform, but it raises some problems if it is not. In a virtualized environment, the kernel is running in a hypervisor-provided virtual address space in much the same way that a userspace process runs in a kernel-provided virtual address space. Allowing the guest kernel to tell devices to write to an arbitrary location in the physical address space is a serious security hole. The situation is even worse if the kernel, or device driver, is not aware that it is running in a virtualized environment. In this case, it could provide an address it believes points to a buffer in the kernel's address space, but that really points somewhere completely different.

In theory, it might be possible for a hypervisor to trap writes to devices and rewrite the DMA addresses to something in the permitted address range. In practice, this is not feasible. Even discounting the (significant) performance penalty that this would incur, detecting a DMA instruction is nontrivial. Each device defines its own protocol for talking to drivers, and so the hypervisor would have to understand this protocol, parse the instruction stream, and perform the substitution. This would end up being more effort than writing the driver in the first place.

On some platforms, it is possible to make use of an *Input/Output Memory Management Unit* (*IOMMU*). This performs a similar feature to a standard MMU; it maps between a physical and a virtual address space. The difference is the application; whereas an MMU performs this mapping for applications running on the CPU, the IOMMU performs it for devices.

The first IOMMU appeared in some early SPARC systems. These came with a network interface that did not have sufficient address space to write into all of main memory. The IOMMU was added to allow pages of the real address space to be mapped to the devices' address space. A different approach was used on x86 platforms when 8- and 16-bit ISA cards were used with 32-bit systems; they simply reserved a block of memory near the bottom of the address space for I/O.

AMD's x86-64 systems also have an IOMMU, for a similar purpose. Many devices connected to x86-64 machines are likely to be legacy PCI devices that only support a 32-bit address space. Without an IOMMU, these are limited to accessing the bottom 4GB of physical memory. The most obvious time this is a problem is when implementing the `mmap` system call, or virtual memory in general. When a page fault occurs, the block device driver can only perform

DMA transfers into the bottom part of physical memory. If the page fault occurs elsewhere, it must use the CPU to write the data, one word at a time, to the correct address, which is very slow.

A similar mechanism has been used in AGP cards for a while. The *Graphics Address Remapping Table* (*GART*) is a simple IOMMU used to allow loading of textures into an AGP graphics card using DMA transfers, and to allow such cards to use main memory easily. It does not, however, do much to address the needs of virtualization, since not all interactions with an AGP or PCIe graphics card pass through the GART. It is primarily used by on-board GPUs to allow the operating system to allocate more memory to graphics than the BIOS did by default.

1.2 Why Virtualize?

The basic motivation for virtualization is the same as that for multitasking operating systems; computers have more processing power than one task needs. The first computers were built to do one task. The second generation was programmable; these computers could do one task, and then do another task. Eventually, the hardware became fast enough that a machine could do one task and still have spare resources. Multitasking made it possible to take advantage of this unused computing power.

A lot of organizations are now finding that they have a lot of servers all doing single tasks, or small clusters of related tasks. Virtualization allows a number of virtual servers to be consolidated into a single physical machine, without losing the security gained by having completely isolated environments. Several Web hosting companies are now making extensive use of virtualization, because it allows them to give each customer his own virtual machine without requiring a physical machine taking up rack space in the data center.

In some cases, the situation is much worse. An organization may need to run two or more servers for a particular task, in case one fails, even though neither is close to full resource usage. Virtualization can help here, because it is relatively easy to migrate virtual machines from one physical computer to another, making it easy to keep redundant virtual server images synchronized across physical machines.

A virtual machine gets certain features, like cloning, at a very low cost. If you are uncertain about whether a patch will break a production system, you can clone that virtual machine, apply the patch, and see what breaks. This is a lot easier than trying to keep a production machine and a test machine in the same state.

Another big advantage is migration. A virtual machine can be migrated to another host if the hardware begins to experience faults, or if an upgrade is scheduled. It can then be migrated back when the original machine is working again.

Power usage also makes virtualization attractive. An idle server still consumes power. Consolidating a number of servers into virtual machines running on a smaller number of hosts can reduce power costs considerably.

Moving away from the server, a virtual machine is more portable than a physical one. You can save the state of a virtual machine onto a USB flash drive, or something like an iPod, and transport it more easily than even a laptop. When you want to use it, just plug it in and restore.

Finally, a virtual machine provides a much greater degree of isolation than a process in an operating system. This makes it possible to create virtual appliances: virtual machines that just provide a single service to a network. A virtual appliance, unlike its physical counterpart, doesn't take up any space, and can be easily duplicated and run on more nodes if it is too heavily loaded (or just allocated more runtime on a large machine).

1.3 The First Virtual Machine

The first machine to fully support virtualization was IBM's VM, which began life as part of the System/360 project. The idea of System/360 (often shortened to S/360) was to provide a stable architecture and upgrade path to IBM customers. A variety of machines was produced with the same basic architecture, so small businesses could buy a minicomputer if that was all they needed, but upgrade to a large mainframe with the same software later.

One key market IBM identified at the time was people wishing to consolidate System/360 machines. A company with a few System/360 minicomputers could save money by upgrading to a single S/360 mainframe, assuming the mainframe could provide the same features. The Model 67 introduced the idea of a self-virtualizing instruction set.

This meant that a Model 67 could be partitioned easily and appear to be a number of (less powerful) versions of itself. It could even be recursively virtualized; each virtual machine could be further partitioned. This made it very easy to migrate from having a collection of minicomputers to having a single mainframe. Each minicomputer would simply be replaced with a virtual machine, which would be administrated in exactly the same way, from a software perspective.

The latest iteration of VM is z/VM, which runs on IBM's zSeries (later rebranded to System z) machines. These can run a variety of operating systems, from old systems for legacy applications to newer systems such as Linux and AIX in a fully virtualized environment, as well as running native VM/CMS applications.

1.4 The Problem of x86

The 80386 CPU was designed with virtualization in mind. One of the design goals was to allow the running of multiple existing DOS applications at once. At the time, DOS was a 16-bit operating system running 16-bit applications on a 16-bit CPU. The 80386 included a virtual 8086 mode, which allowed an operating system to provide an isolated 8086 environment to older programs, including the old real-mode addressing model running on top of protected mode addressing.

Because there were no existing IA32 applications, and it was expected that future operating systems would natively support multitasking, there was no need to add a virtual 80386 mode.

Even without such a mode the processor would be virtualizable if, according to Popek and Goldberg, the set of control sensitive instructions is a subset of the set of privileged instructions. This means that any instruction that modifies the configuration of resources in the system must either be executed in privileged mode, or trap if it isn't. Unfortunately, there is a set of 17 instructions in the x86 instruction set that does not have this property.

Some of the offending instructions have to do with the segmented memory functions of x86. For example, the **LAR** and **LSL** instructions load information about a specified segment. Because these cannot be trapped, there is no way for the hypervisor to rearrange the memory layout without a guest OS finding out. Others, such as **SIDT**, are problematic because they allow the values of certain condition registers to be set, but have no corresponding load instructions. This means that every time they execute they must be trapped and the new value stored elsewhere as well, so it can be restored when the virtual machine is re-activated.

1.5 Some Solutions

Although x86 is difficult to virtualize, it is also a very attractive target, because it is so widespread. For example, virtualizing the Alpha is much easier, however the installed base of Alpha CPUs is insignificant compared to that of x86, giving a much smaller potential market.

Since the IBM PC, x86-based systems have been very popular for business use, leading to a wide selection of legacy business systems. Because of the large potential returns from delivering a working virtualization solution for x86, much effort has been put into getting around the limitations intrinsic to the platform, and a few solutions have been proposed.

1.5.1 Binary Rewriting

One approach, popularized by VMWare, is binary rewriting. This has the nice benefit that it allows most of the virtual environment to run in userspace, but imposes a performance penalty.

The binary rewriting approach requires that the instruction stream be scanned by the virtualization environment and privileged instructions identified. These are then rewritten to point to their emulated versions.

Performance from this approach is not ideal, particularly when doing anything I/O intensive. Aggressive caching of the locations of unsafe instructions can give a speed boost, but this comes at the expense of memory usage. Performance is typically between 80-97% that of the host machine, with worse performance in code segments high in privileged instructions.

There are a few things that make binary rewriting difficult. Some applications, particularly debuggers, inspect the instruction stream themselves. For this reason, virtualization software employing this approach is required to keep the original code in place, rather than simply replacing the invalid instructions.

In implementation, this is actually very similar to how a debugger works. For a debugger to be useful, it must provide the ability to set breakpoints, which will cause the running process to be interrupted and allow it to be inspected by the user. A virtualization environment that uses this technique does something similar. It inserts breakpoints on any jump and on any unsafe instruction. When it gets to a jump, the instruction stream reader needs to quickly scan the next part for unsafe instructions and mark them. When it reaches an unsafe instruction, it has to emulate it.

Pentium and newer machines include a number of features to make implementing a debugger easier. These features allow particular addresses to be marked, for example, and the debugger automatically activated. These can be used when writing a virtual machine that works in this way. Consider the hypothetical instruction stream in Figure 1.1. Here, two breakpoint registers would be used, `DR0` and `DR1`, with values set to 4 and 8, respectively. When the first breakpoint is reached, the system emulates the privileged instruction, sets the program counter to 5, and resumes. When the second is reached, it scans the jump target and sets the debug registers accordingly. Ideally, it caches these values, so the next time it jumps to the same place it can just load the debug register values.

1.5.2 Paravirtualization

The paravirtualization approach involves taking a step back from the problem and modifying the question slightly. Because we cannot easily virtualize x86, paravirtualization asks the question, “What is the closest system to x86 that we can

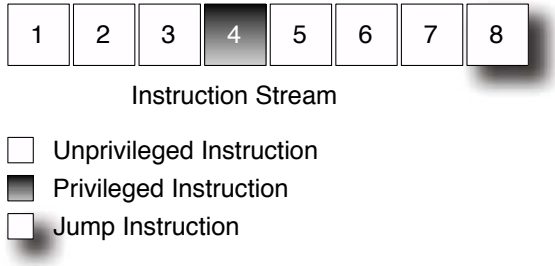

Figure 1.1: An instruction stream in a VM

virtualize?" Rather than dealing with problematic instructions, paravirtualization systems like Xen simply ignore them.

If a guest system executes an instruction that doesn't trap while inside a paravirtualized environment, then the guest has to deal with the consequences. Conceptually, this is similar to the binary rewriting approach, except that the rewriting happens at compile time (or design time), rather than at runtime.

The environment presented to a Xen guest is not quite the same as that of a real x86 system. It is sufficiently similar, however, in that it is usually a fairly simple task to port an operating system to Xen.

From the perspective of an operating system, the biggest difference is that it runs in ring 1 on a Xen system, instead of ring 0. This means that it cannot perform any privileged instructions. In order to provide similar functionality, the hypervisor exposes a set of *hypercalls* that correspond to the instructions.

A hypercall is conceptually similar to a system call. On UNIX[3] systems, the convention for invoking a system call is to push the values and then raise an interrupt, or invoke a system call instruction if one exists. To issue the `exit(0)` system call on FreeBSD, for example, you would execute a sequence of instructions similar to that shown in Listing 1.1.

Listing 1.1: A simple FreeBSD system call

```
1    push    dword 0
2    mov eax, 1
3    push    eax
4    int 80h
```

When interrupt 80h is raised, a kernel interrupt handler is invoked. This reads

[3]Note that Linux uses the MS-DOS system call convention, and so passes parameters in registers.

the value of **EAX** and discovers that it is 1. It then jumps to the handler for this system call, **POP**s the parameters off the stack, and then handle it.

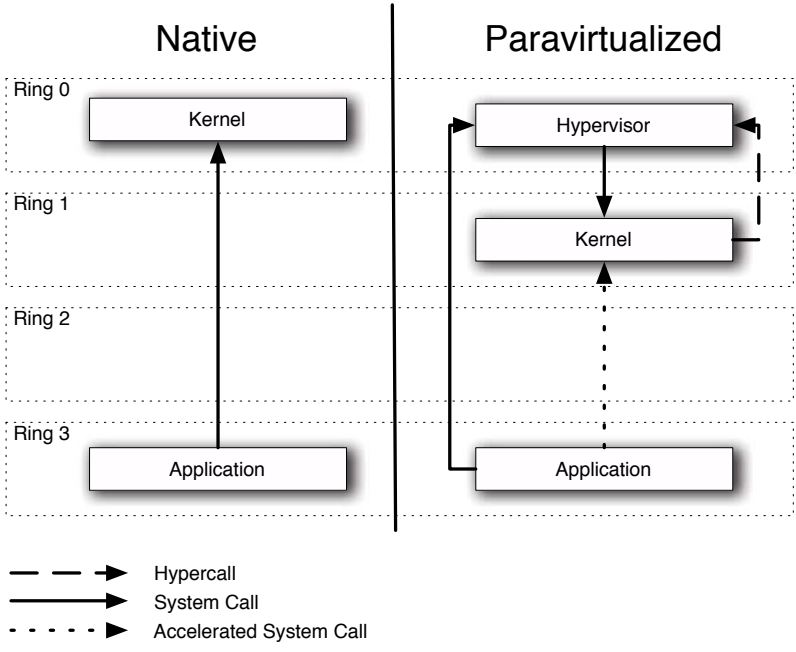

Figure 1.2: System calls in native and paravirtualized systems

Hypercalls work in a very similar manner. The main difference is that they use a different interrupt number (82h, in the case of Xen). Figure 1.2 illustrates the difference, and shows the ring transitions when a system call is issued from an application running in a virtualized OS. Here, the hypervisor, not the kernel, has interrupt handlers installed. Thus, when interrupt 80h is raised, execution jumps to the hypervisor, which then passes control back to the guest OS. This extra layer of indirection imposes a small speed penalty, but it does allow unmodified applications to be run. Xen also provides a mechanism for direct system calls, although these require a modified libc.

Note that Xen, like Linux, uses the MS-DOS calling convention, rather than the UNIX convention used by FreeBSD. This means that parameters for hypercalls are stored in registers, starting at **EBX**, rather than being passed on the stack.

In more recent versions of Xen, hypercalls are issued via an extra layer of indirection. The guest kernel calls a function in a shared memory page (mapped by the hypervisor) with the arguments passed in registers. This allows more efficient mechanisms to be used for hypercalls on systems that support them,

without requiring the guest kernel to be recompiled for every minor variation in architecture. Newer chips from AMD and Intel provide mechanisms for fast transitions to and from ring 0. This layer of indirection allows these to be used when available.

1.5.3 Hardware-Assisted Virtualization

The first x86 chip, the 8086, was a simple 16-bit design, with no memory management unit or hardware floating point capability. Gradually, the processor family has evolved, gaining memory management with the 286, 32-bit extensions, on-chip floating point with the 486 and vector extensions with the Pentium series.

At some points, different manufacturers have extended the architecture in different ways. AMD added 3DNow! vector instructions, while Intel added MMX and SSE. VIA added some extra instructions for cryptography, and enabled page-level memory protection.

Now, both Intel and AMD have added a set of instructions that makes virtualization considerably easier for x86. AMD introduced *AMD-V*, formerly known as *Pacifica*, whereas Intel's extensions are known simply as (Intel) *Virtualization Technology* (*IVT* or *VT*). The idea behind these is to extend the x86 ISA to make up for the shortcomings in the existing instruction set. Conceptually, they can be thought of as adding a "ring -1" above ring 0, allowing the OS to stay where it expects to be and catching attempts to access the hardware directly. In implementation, more than one ring is added, but the important thing is that there is an extra privilege mode where a hypervisor can trap and emulate operations that would previously have silently failed.

IVT adds a new mode to the processor, called *VMX*. A hypervisor can run in VMX mode and be invisible to the operating system, running in ring 0. When the CPU is in VMX mode, it looks normal from the perspective of an unmodified OS. All instructions do what they would be expected to, from the perspective of the guest, and there are no unexpected failures as long as the hypervisor correctly performs the emulation.

A set of extra instructions is added that can be used by a process in VMX root mode. These instructions do things like allocating a memory page on which to store a full copy of the CPU state, start, and stop a VM. Finally, a set of bitmaps is defined indicating whether a particular interrupt, instruction, or exception should be passed to the virtual machine's OS running in ring 0 or by the hypervisor running in VMX root mode.

In addition to the features of Intel's VT[4], AMD's Pacifica provides a few extra things linked to the x86-64 extensions and to the Opteron architecture. Current Opterons have an on-die memory controller. Because of the tight integration

[4]Technically, VT-x for x86. Intel also added similar instructions to Itanium (IA64), known as VT-i.

between the memory controller and the CPU, it is possible for the hypervisor to delegate some of the partitioning to the memory controller.

Using AMD-V, there are two ways in which the hypervisor can handle memory partitioning. In fact, two modes are provided. The first, *Shadow Page Tables*, allows the hypervisor to trap whenever the guest OS attempts to modify its page tables and change the mapping itself. This is done, in simple terms, by marking the page tables as read only, and catching the resulting fault to the hypervisor, instead of the guest operating system kernel. The second mode is a little more complicated. *Nested Page Tables* allow a lot of this to be done in hardware. Nested page tables do exactly what their name implies; they add another layer of indirection to virtual memory. The MMU already handles virtual to physical translations as defined by the OS. Now, these "physical" addresses are translated to real physical addresses using another set of page tables defined by the hypervisor. Because the translation is done in hardware, it is almost as fast as normal virtual memory lookups.

The other additional feature of Pacifica is that it specifies a *Device Exclusion Vector* interface. This masks the addresses that a device is allowed to write to, so a device can only write to a specific guest's address space.

In some cases, hardware virtualization is much faster than doing it in software. In other cases, it can be slower. Programs such as VMWare now use a hybrid approach, where a few things are offloaded to the hardware, but the rest is still done in software.

When compared to paravirtualization, hardware assisted virtualization, often referred to as HVM (Hardware Virtual Machine), offers some trade-offs. It allows the running of unmodified operating systems. This can be particularly useful, because one use for virtualization is running legacy systems for which the source code may not be available. The cost of this is speed and flexibility. An unmodified guest does not know that it is running in a virtual environment, and so can't take advantage of any of the features of virtualization easily. In addition, it is likely to be slower for the same reason.

Nevertheless, it is possible for a paravirtualization system to make some use of HVM features to speed up certain operations. This *hybrid virtualization* approach offers the best of both worlds. Some things are faster for HVM-assisted guests, such as system calls. A guest in an HVM environment can use the accelerated transitions to ring 0 for system calls, because it has not been moved from ring 0 to ring 1. It can also take advantage of hardware support for nested page tables, reducing the number of hypercalls required for virtual memory operations. A paravirtualized guest can often perform I/O more efficiently, because it can use lightweight interfaces to devices, rather than relying on emulated hardware. A hybrid guest combines these advantages.

1.6 The Xen Philosophy

The rest of this book will discuss the Xen system in detail, but in order to understand the details, it is worth taking the time to understand the broad design of Xen. Understanding this, the philosophy of Xen, makes it easier to see why particular design decisions were made, and how all of the parts fit together.

1.6.1 Separation of Policy and Mechanism

One key idea in good system design is that of separation of policy and mechanism, and this is a fundamental part of Xen design. The Xen hypervisor implements mechanisms, but leaves policy up to the Domain 0 guest.

Xen does not support any devices natively. Instead, it provides a mechanism by which a guest operating system can be given direct access to a physical device. The guest OS can then use an existing device driver.

Of course, an existing device driver is not the whole story, because it is unlikely to have been written with virtualization in mind. There also needs to be a way of providing access to the device to more than one guest. Again, Xen provides only a mechanism. The grant table interface allows developers to grant access to memory pages to other guests, in much the same way as POSIX shared memory, whereas the XenStore provides a filesystem-like hierarchy (complete with access control) that can be used to implement discovery of shared pages.

This is not to say that complete anarchy reigns. The Xen hypervisor only implements these basic mechanisms, but guests are required to cooperate if they want to use them; if a device advertises its presence in one part of the XenStore tree, other guests must know to look there if they want to find a device of this type. As such, there are a number of conventions that exist, and some higher-level mechanisms, such as ring buffers, that are used for passing requests and responses between domains for supporting I/O. These are defined by specifications and documentation, however, and not enforced in the code, which makes the Xen system very flexible.

1.6.2 Less Is More

In contrast with most other software packages, each new release of Xen attempts to do less than the previous version. The reason for this is that Xen runs at a very high level of privilege—above even the operating system. A bug in a program may compromise the data that that program can access, a bug in a kernel might compromise an entire system, but a bug in Xen can compromise every virtual machine running on a machine. For this reason, it is important that the Xen code be as secure and bug-free as possible.

To make it easier to audit, the Xen code-base is kept as small as possible. Efficient use of developer time is also important. The Xen developer community is relatively small compared to projects such as Linux (although this may change) and it makes more sense for them to focus on features unique to a hypervisor than duplicate the work of other projects. If Linux already supports a device, then writing a device driver for Xen would be a waste of effort. Instead, Xen delegates device support to existing operating systems.

To maintain flexibility, Xen does not enforce mechanisms for communicating between domains. Instead, it provides simple mechanisms, such as shared memory, and allows guest operating systems to use this as they will. This means that adding support for a new category of device does not require modifying Xen.

Early versions of Xen did a lot more in the hypervisor. Network multiplexing, for example, was part of Xen 1.0, but was later moved into Domain 0. Most operating systems already include very flexible features for bridging and tunnelling virtual network interfaces, so it makes more sense to use these than implement something new.

Another advantage of relying on Domain 0 features is ease of administration. In the case of networks, a tool such as pf or iptables is incredibly complicated, and a BSD or Linux administrator already has a significant amount of time and effort invested in learning about it. Such an administrator can use Xen easily, since she can re-use her existing knowledge.

1.7 The Xen Architecture

Xen sits between the OS and the hardware, and provides a virtual environment in which a kernel can run. The three core components of any system involving Xen are the hypervisor, kernel, and userspace applications. How they all fit together is important. The layering in Xen is not quite absolute; not all guests are created equal, one in particular is significantly more equal than the others.

1.7.1 The Hypervisor, the OS, and the Applications

As mentioned before, one of the biggest changes for a kernel running under Xen is that it has been evicted from ring 0. Where it goes varies from platform to platform. On IA32 systems, it is moved down to ring 1, as shown in Figure 1.3. This allows it to access memory allocated to applications that run in ring 3, but protects it from applications and other kernels. The hypervisor, in ring 0, is protected from kernels in ring 1, and applications in ring 3.

When AMD tidied up the IA32 architecture as part of the process of creating x86-64, one of the things it did was reduce the number of rings. With the exception of OS/2, and (optionally) NetWare, no one at the time made much use of rings 1

and 2, so they wouldn't be missed. Unfortunately, the virtualization community was among those affected.

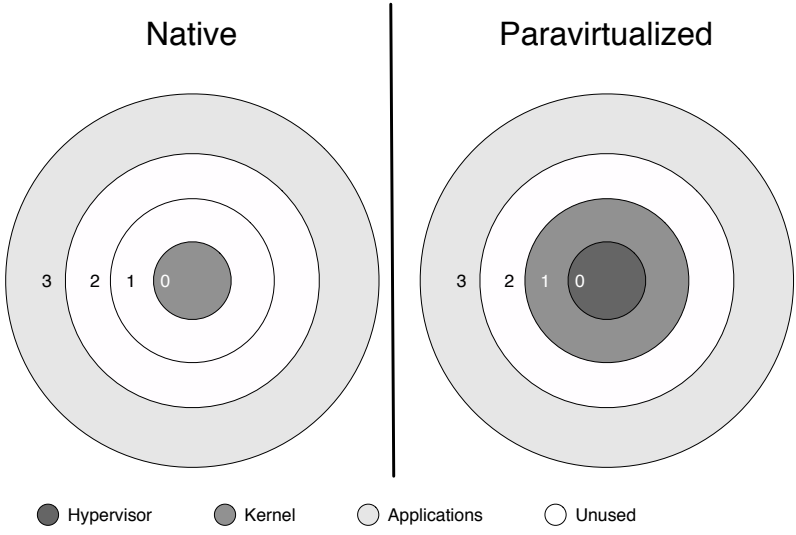

Figure 1.3: Ring usage in native and paravirtualized systems

In the absence of rings 1 and 2, it was necessary to modify Xen to put the operating system in ring 3, along with the applications. Figure 1.4 shows the difference between the two approaches. This approach is also taken by Xen on other platforms, such as IA64, which only have two protection rings. x86-64 also removed segment-based memory protection. This means that Xen has to rely on the paging protection mechanisms to isolate itself from guests.

From the perspective of a paravirtualized kernel, there are quite a few differences between running in Xen and running on the metal. The first is the CPU mode at boot time. All x86 processors since the 8086 have started in *real mode*. For the 8086 and 8088, this was the only mode available; a 16-bit mode with access to a 20-bit address space and no memory management. Since all subsequent x86 machines have been expected to be able to run legacy software, including operating systems, all IBM-compatible PCs have started with the CPU in real mode. One of the first tasks for a modern operating system is to switch the CPU into *protected mode*, which provides some facilities for isolating process memory states, and allows execution of 32-bit instructions.

Because Xen is responsible for system start, it performs this transition itself. If it did not, it would not be able to isolate itself from interference by guest operating

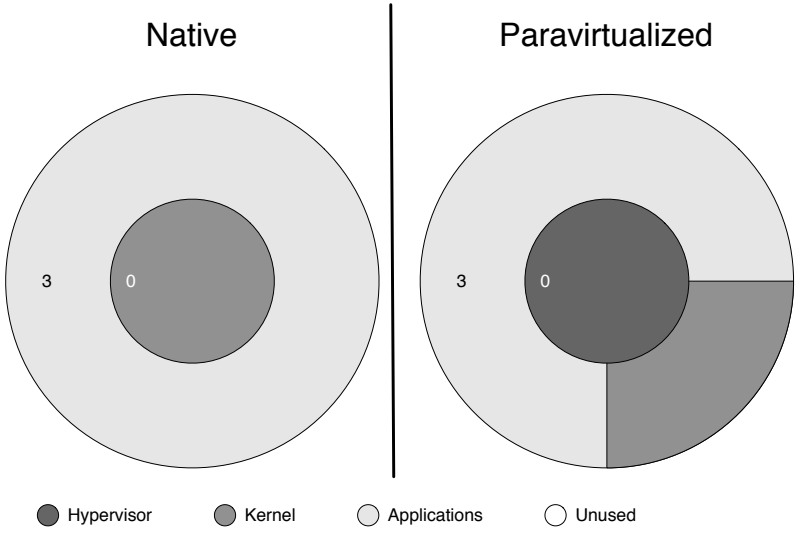

Figure 1.4: Ring usage in x86-64 native and paravirtualized systems

systems. This means that the guest kernel boots in quite a different environment. Newer x86 systems come with Intel's *Extended Firmware Interface* (*EFI*), which is a replacement for the aging PC BIOS. Any system with EFI can also boot in protected mode, although most tend to reuse old boot code and require a BIOS compatibility EFI module to be loaded.

The next obvious change is the fact that privileged instructions must be replaced with hypercalls, as covered earlier. A more obvious change, however, is how time keeping is handled. An operating system needs to keep track of time in two ways: it needs to know the amount of actual time that has elapsed and the amount of CPU time. The first is required for user interfacing, so the user is given a real clock both for display and for programs such as cron, and for synchronizing events across a network. The second is required for multitasking. Each process should get a fair share of the CPU.

When running outside a hypervisor, real time and CPU time are the same thing. All the kernel has to do is keep track of how much time it allocates to running processes and its own threads. When running in Xen, however, it has to share the available CPUs with other operating systems. This is likely to mean that it will only receive some portion of a second of CPU time for every second of real time. As such, it must continually resynchronize its internal clock with the timekeeping facilities provided by Xen.

1.7.2 The Rôle of Domain 0

The purpose of a hypervisor is to allow guests to be run. Xen runs guests in environments known as *domain*s, which encapsulate a complete running virtual environment. When Xen boots, one of the first things it does is load a *Domain 0* (*dom0*) guest kernel. This is typically specified in the boot loader as a module, and so can be loaded without any filesystem drivers being available. Domain 0 is the first guest to run, and has elevated privileges. In contrast, other domains are referred to as *domain U* (*domU*)—the "U" stands for unprivileged. However, it is now possible to delegate some of dom0's responsibilities to domU guests, which blurs this line slightly.

Domain 0 is very important to a Xen system. Xen does not include any device drivers by itself, nor a user interface. These are all provided by the operating system and userspace tools running in the dom0 guest. The Domain 0 guest is typically Linux, although NetBSD and Solaris can also be used and other operating systems such as FreeBSD are likely to add support in the future. Linux is used by most of the Xen developers, and both are distributed under the same conditions—the GNU General Public License.

The most obvious task performed by the dom0 guest is to handle devices. This guest runs at a higher level of privilege than others, and so can access the hardware. For this reason, it is vital that the privileged guest be properly secured.

Part of the responsibility for handling devices is the multiplexing of them for virtual machines. Because most hardware doesn't natively support being accessed by multiple operating systems (yet), it is necessary for some part of the system to provide each guest with its own virtual device.

Figure 1.5 shows what happens to a packet when it is sent by an application running in a domU guest. First, it travels through the TCP/IP stack as it would normally. The bottom of the stack, however, is not a normal network interface driver. It is a simple piece of code that puts the packet into some shared memory. The memory segment has been previously shared using Xen grant tables and advertised via the XenStore.

The other half of the split device driver, running on the dom0 guest, reads the packet from the buffer, and inserts it into the firewalling components of the operating system—typically something like iptables or pf, which routes it as it would a packet coming from a real interface. Once the packet has passed through any relevant firewalling rules, it makes its way down to the real device driver. This is able to write to certain areas of memory reserved for I/O, and may require access to IRQs via Xen. The physical network device then sends the packet.

Note that the split network device here is the same irrespective of the real networking card. Xen provides a simplified interface to these devices, which is easy to implement for people porting systems to Xen. There are three components to any driver:

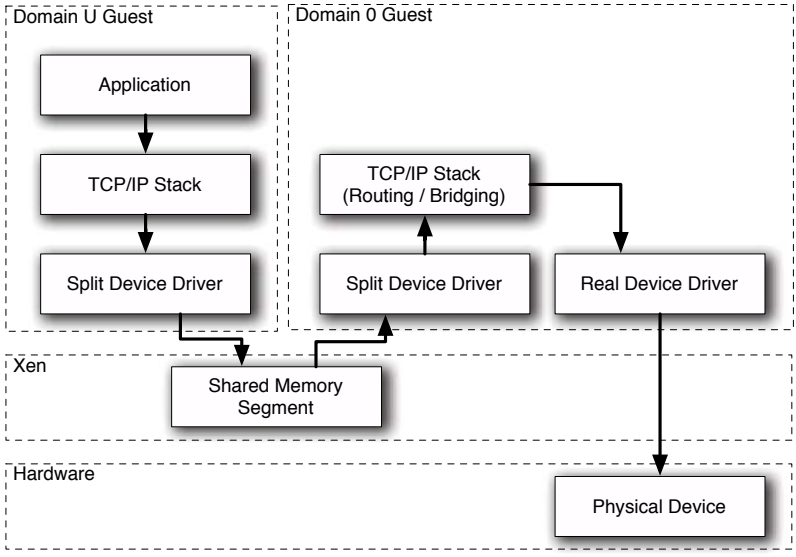

Figure 1.5: The path of a packet sent from an unprivileged guest through the system

- The split driver
- The multiplexer
- The real driver

The split driver is typically as simple as it can be. It is designed to move data from the domU guests to the dom0 guest, usually using ring buffers in shared memory.

The real driver should already exist in the dom0 operating system, and so it cannot really be considered part of Xen. The multiplexer may or may not. In the example of networking, the firewalling component of the network stack already provides this functionality. In others, there may be no existing operating system component that can be pressed into use.

The dom0 guest is also responsible for handling administrative tasks. While Xen itself creates new domU guests, it does so in response to a hypercall from the dom0 guest. This is typically done via a set of Python tools (scripts) that handles all of the policy related to guest creation in userspace and issue the relevant hypercalls.

Domain 0 provides the user interface to the hypervisor. The two dæmons `xend` and `xenstored` running in this domain provide important features for the system.

The first is responsible for providing an administrative interface to the hypervisor, allowing a user to define policy. The second provides the back-end storage for the XenStore.

1.7.3 Unprivileged Domains

An *unprivileged domain* (*domU*) guest is more restricted. A domU guest is typically not allowed to perform any hypercalls that directly access hardware, although in some situations it may be granted access to one or more devices.

Instead of directly accessing hardware, a domU guest typically implements the front end of some split device drivers. At a minimum, it is likely to need the XenStore and console device drivers. Most also implement the block and network interfaces. Because these are generic, abstract, devices, a domain U guest only needs to implement one driver for each device category. For this reason, there have been a number of operating systems ported to run as domain U Xen guests, which have relatively poor hardware support when running directly on real hardware. Xen allows these guests to take advantage of the Domain 0 guest's hardware support.

Unlike dom0 guests, you can have an arbitrary number of domU guests on a single machine, and they may be able to be migrated. Whether migration is possible depends largely on the configuration; guests configured with tight coupling to hardware cannot be moved. A guest that uses something like NFS or iSCSI for all of its storage requirements can be migrated live, whereas one that uses a block device driver could be suspended to a flash drive and moved to a different machine, as long as it does not depend on any other hardware that does not support migration yet.

For security reasons, it is advisable to do as little in Domain 0 as possible. A root exploit in Domain 0 could potentially compromise the entire system. As such, most work should be done in paravirtualized domain U guests or HVM guests (discussed in the next section).

The line between dom0 and domU is sometimes blurred. It is possible to allow a domU guest direct access to some hardware, and even to host split device drivers in one. For example, a laptop might use Linux as the dom0 guest, but run a NetBSD domU VM in order to support a particular WiFi card. On platforms without an IOMMU, doing this can compromise security, because it potentially allows the domU guest access to the entire address space.

1.7.4 HVM Domains

When Xen was created, the x86 architecture did not meet Popek and Goldberg's requirements for virtualization. A virtual machine monitor for x86 needed to emulate the architecture, although it could do it very quickly for a large subset of instructions. Xen implemented naravirtualization to avoid this problem.

More recent x86 chips have not suffered from this limitation, and so it made sense to extend Xen to support unmodified guests. Most of the discussion in this chapter has related to paravirtualized guests, which, until the 3.x series, were

the only kind supported by Xen. More recent versions, however, also allow the running of *Hardware Virtual Machine* (*HVM*) guests.

Running unmodified guests in Xen requires hardware support as described earlier, and so is not an option for older machines. Any x86 system bought in 2007 or later should support HVM, and some machines from 2006 will.

An unmodified OS is not likely to have support for Xen split device drivers (although if a driver development kit is available, it may be possible to implement them). This means that Xen must emulate something that the guest is likely to support. A small number of devices are available in this way, using code from QEMU.

HVM guests differ from paravirtualized guests in a number of ways. This is apparent from boot time. A paravirtualized guest begins in protected mode, with some memory pages containing boot information mapped by the hypervisor, whereas an HVM guest begins in real mode and gets configuration information from an emulated BIOS.

If an HVM guest wants to take advantage of Xen-specific features, it needs to use the **CPUID** instruction to access a (virtual) machine specific register and access the hypercall page. It can then issue hypercalls in the same way as a paravirtualized guest, by calling an offset in the hypercall page. This then uses the correct instruction (e.g., **VMCALL**) for a fast transition to the hypervisor.

For virtualization-aware HVM guests, much of the information otherwise available is accessed via the *platform PCI device*, a virtual PCI device that exports hypervisor functionality to the guest.

1.7.5 Xen Configurations

The simplest possible configuration for Xen is to run nothing other than a single dom0 guest. Here, Xen does little other than act as a simple hardware abstraction layer, hiding some of the messier parts of the x86 architecture from the kernel. This configuration is not very useful, however.

In practice, a Xen system typically has at least one domU guest. The simplest useful configuration would look something like Figure 1.6.

In this example, all of the hardware is being controlled by the host in Domain 0. This is not always ideal. If a driver for a particular device contains bugs, it can crash the dom0 kernel, which in turn can bring down all guests on the system. Therefore, it is often beneficial to isolate a driver in its own domain, which does nothing other than export the split device driver. This can either be a full-fledged operating system, such as Linux or a BSD variant, or a driver housed in a lightweight OS, such as the miniOS example included with Xen. On platforms with an IOMMU, this can completely prevent the buggy driver from interfering with other guests. On systems without the benefit of an IOMMU, the remaining

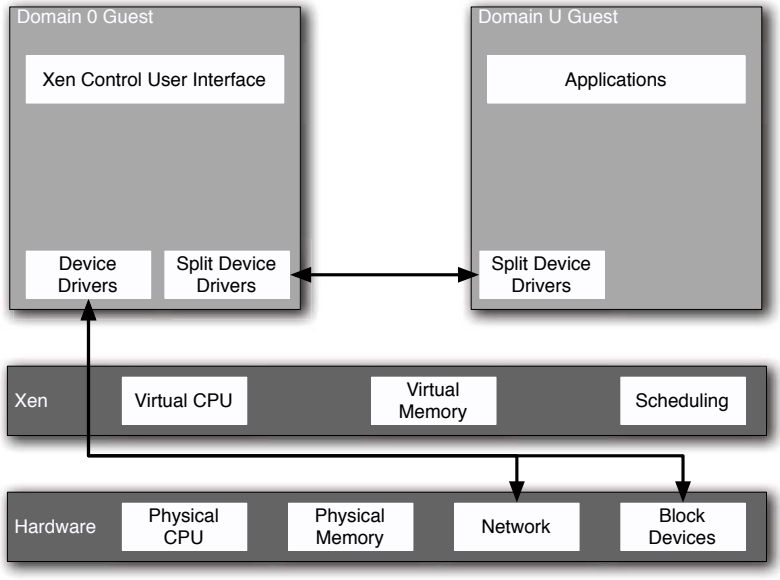

Figure 1.6: A simple Xen configuration

guests are still vulnerable to erroneous DMA requests and possible I/O outages, but are isolated from other bugs.

When Xen is used for backward-compatibility by running a legacy operating system concurrently with a new system, it is likely that running an unmodified guest is going to be required. In this situation, unmodified guests will not use the split device drivers directly. Instead, they will use emulated devices run from Domain 0. This configuration is shown in Figure 1.7.

Finally, in a clustering environment, it might be useful to be able to dynamically redistribute guests depending on the current load on the system. To do this, a typical configuration might have a single file server mounted over NFS for each of the clients, and a single router handling external routing.[5] In this situation, no guest would depend on the local block device driver, and could be migrated between nodes easily. This configuration is shown in Figure 1.8. Note that the individual nodes do not even possess a local block device, simplifying cluster costs. In this situation, it is likely that Xen and the Domain 0 guest would be booted

[5]In a high-availability environment, it is possible to add redundancy to these, for example, by using automatic fail-over for the firewall and backup NFS servers connected to the same SAN.

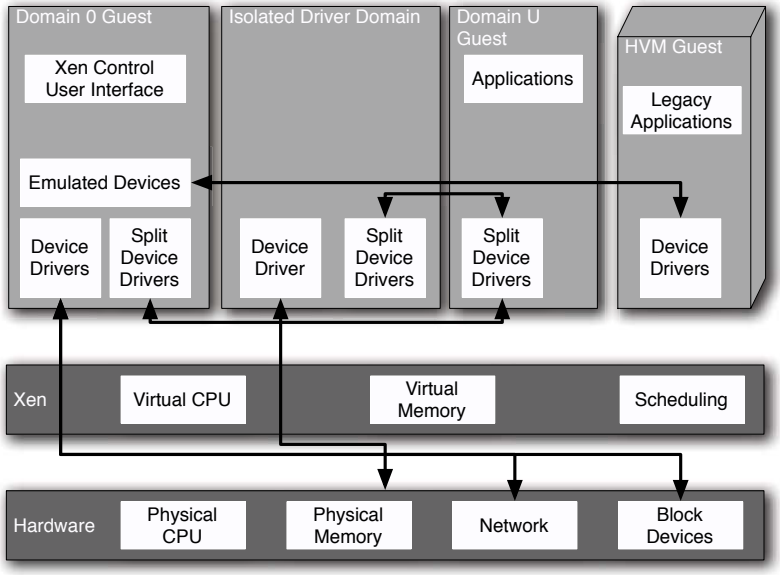

Figure 1.7: A Xen configuration showing driver isolation and an unmodified guest OS

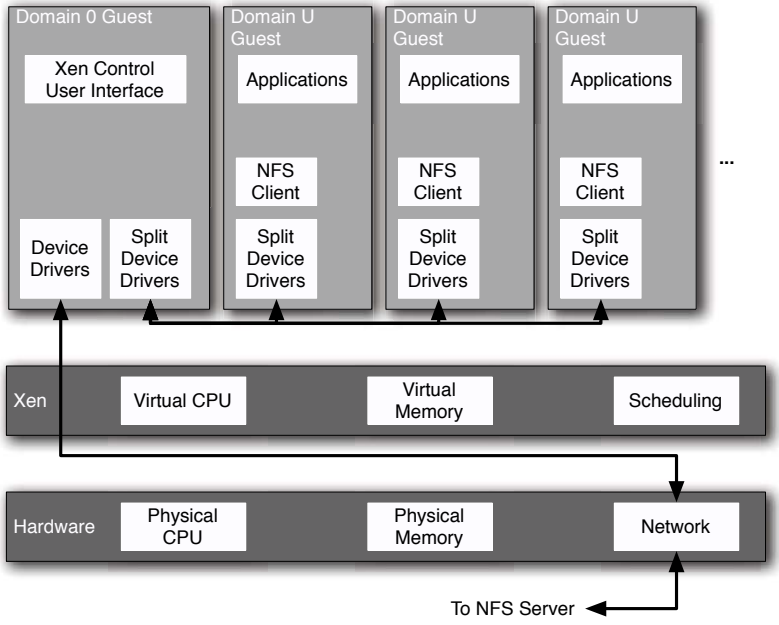

Figure 1.8: A single node in a clustered Xen environment

using PXE, or some equivalent, allowing individual cluster nodes to be replaced whenever they failed.

Migration in Xen causes very small amounts of downtime. In tests, a Quake server was migrated among nodes in a cluster without the connected players noticing.

There are a lot of different possible configurations for a Xen system, for a lot of potential uses. It can aid debugging, testing, running legacy software, running reliable systems, and dynamic load balancing. One of the earliest uses for virtualization was consolidating various different minicomputer workloads onto a single mainframe. More modern versions of this are still common, with virtual machines being used to isolate independent workloads on the same machine. Often, these workloads may be running the same operating system, with virtualization being used to allow different individuals to administer their own workloads, or simply to provide logical partitioning of the system.

The consolidation that this approach offers gives rise to a much greater degree of efficiency than having large numbers of individual machines. This gives cost savings in terms of required space, power, and cooling, as well as the initial hardware investment.

Chapter 2

Exploring the Xen Virtual Architecture

This chapter will examine the main features of Xen. The last chapter looked at some of the reasons for virtualizing x86, why it is hard, and gave a very brief overview of how Xen addresses some of these problems. We will now take a more detailed look at how the various parts of Xen interact with each other, and what each part does. We will also examine how a paravirtualized Xen guest environment differs from real hardware.

2.1 Booting as a Paravirtualized Guest

When an x86 processor is started, it begins in real mode where it is compatible with the 8088 found in the original IBM PC. One of the first things a modern operating system does is enter protected mode and install interrupt handlers. Xen is no exception; it enters protected mode and installs its own interrupt handlers. This means that when an operating system boots inside a Xen virtual machine, the CPU is already in protected mode.

Although Xen's interrupt handlers are installed, most interrupts will not be properly handled by Xen. Instead, they will be passed on to any guest that requests them. Some forms of interrupt relate to CPU state, and will be bounced back to the currently running guest. Others are used for communicating with hardware and will be turned into Xen events and delivered via an upcall into the domain containing the driver.

A multitasking operating system, at the very least, needs to register for the clock interrupt to know when to switch execution to the next task. Typically, it also registers some interrupts related to hardware devices, such as the keyboard.

Note that Xen needs to use the timer interrupt for scheduling guests, as well as generating events in running guests for its own scheduling.

On most architectures, the correct way of installing interrupt handlers is to write a jump address into a part of memory that corresponds to the interrupt number, known as the *interrupt vector*. On Xen systems, hypercalls bind an event port to a virtual or physical IRQ. Some things that must be handled immediately (mainly CPU exceptions) are dispatched immediately to a running guest, if handlers have been defined. Others, such as a notification that there is data available from a particular virtual device might be less immediate.

A physical guest typically spends some of its boot time firing off queries to the BIOS to determine what hardware is available, including CPU capabilities. In a Xen guest, however, the BIOS is unavailable. Because the BIOS allows direct access to the hardware, permitting guests to interact with it directly would break the principle of isolation.

The BIOS is replaced by a variety of different facilities in Xen. The first is the start info page, which contains basic information required by a guest to initialize the kernel. Next is the shared info page, which gives some more data and is updated while the guest is run. Finally, there is the XenStore. Among other things, this is used to determine which (virtual) devices are available.

The devices available to a Xen guest are not real devices. Instead, they are abstracted virtual devices. This means that a (domU) Xen guest only needs to supply a single driver for each category of device it wants to support. It also removes some of the overhead incurred by full-virtualization systems supporting emulated hardware. Where real devices are emulated, the guest OS has to turn abstract requests into device-specific commands. The emulator has to turn these back into an abstract form, and the underlying driver has to turn these into commands for the real device. Xen's device architecture attempts to keep the commands in an abstract form so only a small amount of processing time is required to get them from the guest to the driver for the physical device.

2.2 Restricting Operations with Privilege Rings

The x86 architecture is quite unusual in the way it handles privileged instructions. Most architectures have two modes, privileged and unprivileged, and some instructions may only be used from privileged mode. The system Intel created for the 80386 was somewhat more flexible, providing four privilege levels, known as rings. This design was similar to that of DEC's VAX, a popular architecture at the time of the design of the 80386. VMS, the native operating system for the VAX, made use of all four rings, and so it was believed that this would aid porting of VMS, and would be useful to designers of new operating systems. Since then,

VMS, which is now sold by HP, has been ported to the Alpha and Itanium (both of which only support two rings), but not x86.

The concept of multiple hardware-enforced privilege layers originated in MULTICS, which proposed a system of eight rings. The UNIX system was designed to run on simpler systems (the DEC PDP-11) and so only required two: one for the kernel and one for everyone else.

Most operating systems written for IA32, including Windows NT and most UNIX-like systems, only use two. One exception to this was OS/2, which ran device drivers in a lower privilege ring than the rest of the kernel. Another was Novell Netware, which, in later versions, loaded modules in lower privileged rings. As with many of the other unusual features of x86, it is only possible to efficiently use the four ring architecture if you sacrifice portability. Because both Windows NT and UNIX began life on non-x86 architecture, they could not do this. Windows NT was first developed on the Intel i860, and later on MIPS, before being ported to x86.

Because most operating systems only use rings 0 and 3, this leaves two spare. Unfortunately, rings 1 and 2 are not very useful as a home for a hypervisor. Because the hypervisor needs to run at a higher privilege level than a kernel, you would ideally put it in at ring -1. Xen does the next best thing: It takes over ring 0, and runs the kernel in ring 1.

This is not always possible. Some non-x86 platforms (including x86-64) only provide two protection levels. In these cases, the hypervisor lives in the privileged mode and both the kernel and userspace applications are moved out to the same unprivileged level. Fortunately, most of the architectures with only two privilege modes support some form of hardware virtualization, and therefore include a virtual ring -1. This allows the hypervisor to live in a higher privilege level (and prevent the kernel from interfering with other running guests) without introducing the complications involved with throwing the kernel out of its accustomed home.

From a practical perspective, this eviction means that a kernel running under Xen can not execute certain privileged operations, without generating a general protection fault. Due to some limitations of the x86 instruction set, a few things will simply silently fail. It is the responsibility of a running guest to ensure that it does not use any of these operations, and instead replaces them with equivalent hypercalls. In the case of guest kernels running in a hardware-assisted virtualization environment, it is still generally preferable from an efficiency perspective to explicitly execute the hypercalls, rather than rely on the CPU trapping privileged instructions and the hypervisor emulating them.

These restrictions only apply to paravirtualized guests. Guests running in *Hardware Virtual Machine* (*HVM*) domains are presented with the appearance of a normal x86 machine. Time keeping is accomplished via an emulated real time clock, and privileged operations are trapped and emulated by the hypervisor. The HVM domain has a set of emulated devices, and so can use existing drivers.

An HVM guest runs in a virtual ring 0, and has memory accesses translated transparently for it. If it is written to be aware of virtualization, it can use some hypervisor-specific functionality usually associated with paravirtualized guests.

2.3 Replacing Privileged Instructions with Hypercalls

Because the kernel is now running in ring 1 where it is not allowed to do everything it wants, there has to be some mechanism for bypassing this restriction in a controlled manner.

This is not a new problem. Userspace code, running in ring 3, encounters it all the time. Displaying things on the screen, reading input from the keyboard, and sending data over the network all involve interaction with hardware, which can't be performed by unprivileged code.

The solution is to use a system call, a formal mechanism for telling the kernel to do something for you. System calls all work in roughly the same way, irrespective of your operating system[1] or platform:

1. Marshal the arguments in registers or on the stack.
2. Issue a well-known interrupt, or invoke a special system call instruction.
3. Jump to the kernel's (privileged) interrupt handler as a result of the interrupt.
4. Process the system call at the kernel's privilege.
5. Drop to a lower privilege level and return.

On x86 it is common to use interrupt 80h for system calls, although newer CPUs have SYSENTER/SYSEXIT or SYSCALL/SYSRET instructions for fast system calls, depending on the manufacturer.

The same mechanism was previously used by Xen. Hypercalls were generated by a guest kernel in almost the same way as system calls are generated by userspace applications, the difference being that interrupt 82h, instead of 80h, is used.

This still works as of Xen 3, but is now deprecated. Instead, hypercalls are issued indirectly via the *hypercall page*. This is a memory page mapped in to the guest's address space when the system is started.

Hypercalls are issued by **CALL**ing an address within this page. Listing 2.1 shows a macro that is used to issue a one-argument hypercall.

Line 5 of this contains some syntactic sugar using the C preprocessor to replace the hypercall name with a macro that is then expanded to the hypercall number.

[1] Some microkernel operating systems replace this with a message passing mechanism; however, the messages tend to be sent using a mechanism similar to that described here.

Fast System Calls

You may have realized that the extra layer of indirection added by Xen adds some overhead to invoking system calls. Rather than issuing an interrupt and it being caught by the kernel's interrupt handler, the interrupt will be caught by Xen's interrupt handler and then passed back to the kernel as an event. This adds another two context switches (application to Xen and Xen to kernel), which is expensive on an architecture like x86 where context switching is not particularly fast.

To address this, Xen provides a fast system call interface. The guest kernel may specify a handler for interrupt 80h, which is then installed by Xen when the guest is running, bypassing the normal Xen handlers.

This is only required on x86 if maintaining compatibility with existing user-land applications is required. If it is not, it is possible to use call gates to handle the transition between ring 3 and ring 1 directly, without involving the hypervisor. System calls can then be exposed as function calls accessed using the **CALL FAR** instruction. Unfortunately, this mechanism is not available in x86-64, and due to lack of use has not been heavily optimized in the design of recent x86 CPUs.

This line calls the address at 32 times the hypercall number offset from the start of the hypercall page. Because the page size on x86 is four kilobytes, this gives a maximum of 128 hypercalls. At the time of writing there are currently 37 hypercalls defined with a further seven reserved for architecture-specific hypercalls, giving some room for future growth.

For comparison, the Version 7 UNIX kernel supported 64 system calls (although not all were implemented) and a modern FreeBSD kernel currently implements around 450 system calls. It may seem odd that Xen has fewer hypercalls than a 1979 release of UNIX had system calls. This is an active example of the Xen philosophy of less is more; the hypervisor should not do anything that it doesn't absolutely have to in order for guests to function. It should be noted that this is not an entirely fair comparison. Several of the Xen hypercalls perform clusters of operations, such as performing scheduler operations or updating the page table. Equivalents would be performed by several different system calls on a *NIX system.

The next two lines tell GCC about the register usage of the call. The return value (`__res`) is stored in **EAX** and the argument passed in **EBX**. For hypercalls with more arguments, **ECX** etc. are used (on x86). The registers used for arguments are clobbered, and certain hypercalls may make changes to the guest's address space or modify structures passed by reference, so memory is treated as clobbered

Listing 2.1: A macro issuing a one-argument new-style hypercall

```
#define _hypercall1(type, name, a1)                 \
({                                                  \
    long __res, __ign1;                             \
    asm volatile (                                  \
        "call hypercall_page + ("STR(__HYPERVISOR_##name)" * 32)"\
        : "=a" (__res), "=b" (__ign1)               \
        : "1" ((long)(a1))                          \
        : "memory" );                               \
    (type)__res;                                    \
})
```

as well. The value of **EBX** is stored in a temporary value and discarded to let the compiler know that **EBX** is clobbered. This must be done this way because GCC inline assembly does not permit input arguments to be in the explicit clobbered list. The value of **EAX** is finally cast to the type specified with the macro.

When you are writing userspace code, it is very rare to have to do the messy steps involved in entering kernelspace manually. Even in something like a C library that makes system calls directly, they are usually wrapped up in a macro. The same is true of hypercalls. A series of macros such as the one described earlier are defined, which makes invoking hypercalls easy. These are then used to build inline functions for common hypercalls.

Listing 2.2 shows an example of such a macro. This particular function issues a hypercall related to scheduling, which takes two arguments. When writing guest kernel code, most privileged operations are performed using functions like this. These macros are defined in a header file which, unfortunately, is not in the public part of the tree. This means that it must be copied by any guest wanting to use Xen. One copy can be found in `extras/minios/include/x86/x86_32/hypercall-x86_32.h` (replace 32 with 64 for the x86-64 version). This file, as with the other Xen interface files, is under a liberal BSD-style license, and so it can likely be included in any project without legal concerns.

Listing 2.2: An example hypercall inline function

```
static inline int
HYPERVISOR_sched_op(
    int cmd, unsigned long arg)
{
    return _hypercall2(int, sched_op, cmd, arg);
}
```

A Xen guest uses the HYPERVISOR_sched_op function with SCHEDOP_yield argument[2], instead of issuing the hlt instruction, in order to relinquish CPU time to guests with running tasks. This is an important example, because it highlights the fact that instructions may have different meanings in a virtual environment; the physical CPU should only be put into a low power mode if all of the guests are idling, not simply if one decides that it doesn't have anything important to do.

2.4 Exploring the Xen Event Model

Writing code for Xen is, in many ways, similar to writing code for UNIX. Table 2.1 shows some of the Xen equivalents for common UNIX interfaces. In a UNIX system it is possible to set up handlers for signals. These are a simple asynchronous mechanism that allows something outside the program to deliver a single bit of data saying "event n has occurred." The external entity can be either the kernel or another application.

The Xen analog is the events mechanism. One of the first things that a guest kernel should do is register a callback to be used for event delivery. When events are delivered to the guest, various flags are set to indicate which event is present. If event delivery is masked, these flags can be used later to check for waiting messages. If not, the event will be delivered asynchronously.

Events can come from Xen directly, and may represent hardware or virtual interrupts, or may be raised by other guests. As with UNIX signals, they can be used to build more complex asynchronous communication paths, for example by allocating an event channel to indicate that the contents of a shared memory page have been updated.

Just like a signal, a Xen event is delivered via a callback. All upcalls are delivered to the same address,[3] which can then call the relevant part of the guest code. This is similar to the mechanism used for UNIX signals in some POSIX-like microkernel operating systems, such as GNU HURD, where signals are delivered via a message port to the userland application which then handles the vectoring to the correct handler itself.

Xen events share a lot in common with hardware interrupts and with UNIX signals, but they are conceptually a closer relative of Mach Ports. Xen events are delivered via channels; you cannot receive arbitrary events from an arbitrary sender. When a UNIX process wants to send a signal to another, it must first discover the remote process ID and then it simply needs to use the kill system

[2] This is the guest kernel equivalent of calling sleep(0) in a UNIX program.

[3] Actually, there are two callbacks registered for event delivery, but the second is only used if a fault occurs.

Table 2.1: Xen components and their UNIX counterparts

UNIX	Xen
System Calls	Hypercalls
Signals	Events
Filesystem	XenStore
POSIX Shared Memory	Grant Tables

call. The equivalent operation in Xen is one guest sending an asynchronous event to another domain. This requires the following steps:

1. Receiving guest creates a new, unbound port.
2. Receiving guest advertises the existence of the port (typically via the XenStore).
3. The sending guest creates a new port if it doesn't have any free.
4. The sending guest binds its port to the remote one.
5. The sending guest sends the event.

On a UNIX system, it is possible for one process to send another one a SIG_SEGV (for example), causing it to crash. This is fine on UNIX where the security model is based around users, so the worst thing that can happen is that a user crashes his own applications. It would not work for Xen, however, because isolation of domains is vital to secure virtualization. This mechanism ensures that a guest kernel will only receive events that it knows how to handle.

2.5 Communicating with Shared Memory

The UNIX signal model is good for delivering events quickly, but it is not enough to build a general purpose *interprocess communication* (*IPC*) mechanism. Other mechanisms, such as pipes, message queues, and shared memory exist for this purpose. Because it is possible to implement mechanisms like pipes and message queues solely from shared memory, it is not necessary for Xen—a minimalist hypervisor—to provide anything except *shared memory*.

Xen supports two basic interdomain operations on memory pages: sharing and transferring. A shared page is similar to a page shared with POSIX shared memory; both domains are able to access the contents. Page transfer is a coarse-grained message-passing mechanism.

Experimental results showed that transferring pages was not as useful as it might at first appear. Early implementations of the network device driver, for example, used the transfer mechanism to transfer packet buffers between domains. It turned out that the page table operations required for this were so expensive that they overshadowed any performance gain that might have been had. Newer versions of Xen implement a hypervisor-driven copy operation instead. Because the hypervisor has all physical memory mapped, this allows data to be copied between domains without modifying any page tables, which ends up being much faster.

The most obvious difference between Xen grant tables and POSIX shared memory is the fact that Xen deals with page tables directly. There is no abstraction presenting the idea of a flat, byte-granularity, address space. A Xen guest can only manipulate shared memory on a page level.

The other significant difference is the way in which pages are identified. With System V shared memory, shared memory regions are identified by an opaque (scalar) type, but with POSIX they exist somewhere in the filesystem. Xen does not have the concept of a filesystem, although the XenStore is similar. Shared memory regions in Xen fall somewhere between the System V and POSIX models. The region itself is identified by a *grant reference*, which is an integer (as with System V). The value of this integer is communicated between domains using the XenStore, in a manner reminiscent of how POSIX uses the filesystem.

It is, of course, possible to pass a grant reference using some mechanism other than the XenStore—for example, sockets over an existing interface. Pages used to transfer data to virtual device drivers will typically be in the `/local/domain/0/backend/` part of the XenStore, but there is nothing stopping cooperating domains from sharing additional pages. This might be used to implement a fast network connection similar to the loopback interface in most operating systems, or to support the MIT shared memory extension for X11, allowing faster displays.

2.6 Split Device Driver Model

The Xen driver model is a good example of the Xen philosophy. Adding support for the myriad devices available for the PC to Xen would be a lot of work (and thus potential for bugs) and a duplication of effort. Instead, Xen delegates hardware support to a guest. This is typically Domain 0, although it is possible to delegate hardware to guests in other domains.

Xen device drivers typically consist of four main components:

- The real driver
- The bottom half of the split driver

- The shared ring buffer(s)
- The top half of the split driver

The real driver is any driver that exists in a current operating system. When the kernel is ported to Xen, things like interrupt handling must be modified to use Xen events, but in a well-designed kernel these are handled by an abstraction layer, so drivers do not need to be modified.

The bottom half of the split driver has two main features: It handles multiplexing and provides a generic interface. A block device, for example, should have a simple set of operations such as read block and write block, which are independent of the real hardware. The multiplexing is to allow more than one guest to use the device. Most devices don't support this natively, but other operating system features might already provide this service. One of the main uses for an operating system (as opposed to running applications directly on the hardware) is to multiplex access to devices. Access to hard disks is multiplexed using a filesystem abstraction, network devices using a socket abstraction, and so on.

Often, the multiplexing provided for applications is too high-level to be useful to other guests. A filesystem abstraction is fine, because each file (on UNIX-like systems) is effectively a virtual block device. For networks, the same is not true. Exposing a socket interface to guests would require them to rewrite a large portion of their networking stack. Fortunately, many operating systems provide lower-level services for bridging, routing, and virtual interfaces that can be used for some of this. A Xen network driver can be added to the Domain 0 kernel as a virtual interface and then existing routing code can be used to handle multiplexing.

When the bottom half supports the necessary features, there needs to be some mechanism for exporting it to other guests. This is typically done using ring buffers in shared memory segments. The front-end driver initializes a memory page with a ring data structure and exports it via the grant table mechanism. It then advertises the grant reference via the XenStore where the back end can retrieve it. The back end then maps it into its own address space, giving a shared communication channel into which the front end inserts requests and the back end places responses. Typically, an event channel is also configured for the device and used by both ends to signal when data is waiting in the ring.

The top half of the driver, running in an unprivileged guest, is typically very simple. It needs to inspect the XenStore to find the address of the shared memory page and map it. Most Xen drivers use a *ring buffer* abstraction for communication. The top half writes commands into the ring while the bottom half writes replies. An event channel may also be used to signal that there is waiting data in the ring.

The ring buffer is located in a shared memory segment. For it to be usable, the Domain 0 guest stores the associated grant table reference in the XenStore. Other guests are then able to enumerate the available devices by reading the XenStore.

Device drivers use most of the features of Xen—grant tables, event channels and the XenStore—to function.

2.7 The VM Lifecycle

The lifecycle of a real machine used to be fairly simple. It would be turned on, would run, and would then be turned off. More recent machines have added one or more suspend states to this, as shown in Figure 2.1. The suspend state in this graph might represent multiple different levels of suspension—for example, a suspend-to-RAM and a deeper suspend-to-disk mode.

This becomes slightly more complicated when we start talking about virtual machines. A virtual machine, as shown in Figure 2.2, has an additional state: paused. This is somewhere between running and suspended. The machine is still resident, but it is never allocated CPU time. When a virtual machine is suspended, the whole thing is serialized to persistent storage, and it is unloaded from memory. The only difference between being suspended and turned off, from the perspective of a virtual machine, is the state that is stored. In both cases, the state of block devices is preserved, but when suspended the contents of memory and the CPU are also preserved. It could be argued that this is also true when the machine is turned off, although the state of the CPU will be the initial boot state, and the state of memory will be "undefined."

The other big change is the extra transition from the running state to the running state. By definition, a real machine is tied to the hardware. A virtual machine can be migrated from one machine to another. This is stated explicitly in the transition diagram, but it can also happen implicitly. If the machine is suspended on one machine, it could be resumed on another.

The manner in which the transitions are accomplished also differs between

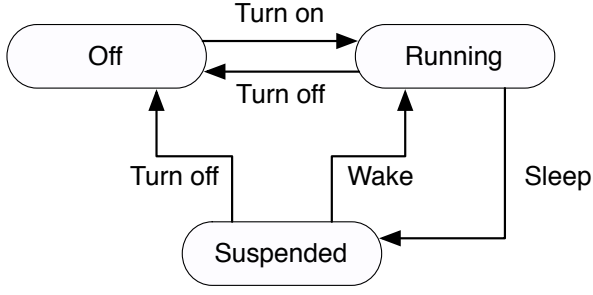

Figure 2.1: The lifecycle of a real machine

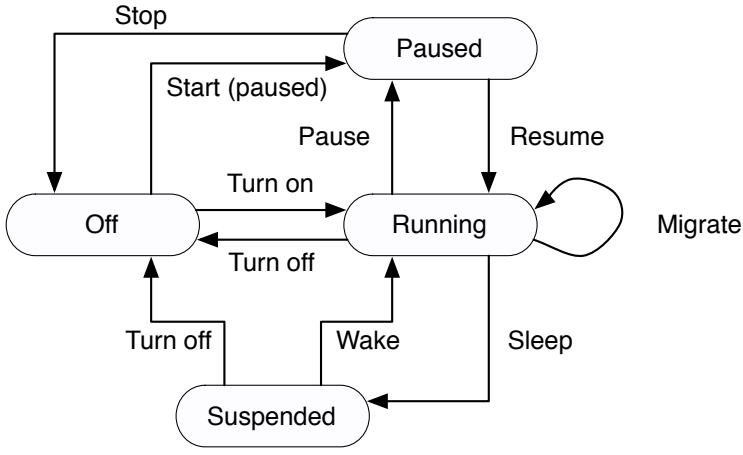

Figure 2.2: The lifecycle of a virtual machine

real and virtual machines. A physical machine is powered on by pressing the on button (or using a mechanism like wake-on-LAN). A Xen guest is started using the management tools, such as `xm` or a web interface to `xend`.

2.8 Exercise: The Simplest Xen Kernel

When you write code for any system, you generally have to write some simple boiler-plate code. Any C program, for example, tends to include a set of standard headers, and a main() function.

Xen is no exception to this. Even a kernel that does nothing requires some extra headers to tell the Xen loader how to launch it. Each Xen guest kernel needs to begin with a __xen_guest section, which contains a NULL-terminated string of key-value pairs for the loader to read.

Listing 2.3: Header for a simple kernel [from: examples/chapter2/bootstrap.x86_32.S]

```
4  .section __xen_guest
5      .ascii  "GUEST_OS=Hacking_Xen_Example"
6      .ascii  ",XEN_VER=xen-3.0"
7      .ascii  ",VIRT_BASE=0x0"
8      .ascii  ",ELF_PADDR_OFFSET=0x0"
9      .ascii  ",HYPERCALL_PAGE=0x2"
10     .ascii  ",PAE=no"
11     .ascii  ",LOADER=generic"
```

```
12      .byte   0
```

Listing 2.3 shows an example __xen_guest section. Note that this is from an x86 kernel; x86-64 headers look similar, although PAE is only relevant to 32-bit x86 systems, so it is not present.

The first key gives the kernel a name. This can be anything you like, and is simply used to identify running kernels. The second option specifies the version of Xen for which the guest was written. This allows the Xen loader to tell whether it can expect to run the kernel.

The next two parameters deal with the guest's address space. When a guest is launched, it has a fixed size memory allocation. This mirrors a kernel running on a physical machine, which has a fixed amount of physical memory.[4] The value of VIRT_BASE determines where in the guest's address space this allocation is mapped. The ELF_PADDR_OFFSET address is the value subtracted from addresses in ELF headers to generate a corresponding address in the guest kernel's address space. A kernel typically wants to be mapped in to the bottom of its address space, so it sets both of these to zero.

The HYPERCALL_PAGE value refers to the new method for issuing hypercalls. The Xen loader maps a page into the guest's address space containing hypercall trampolines. The guest may determine where it wants to have this page mapped. Note that this value is the page number, not the memory address. With 4KB pages, the page 0x2 corresponds to the address 0x2000.

The LOADER section allows you to specify special boot loaders. Currently, "generic" is the only option. Options passed to the loader are also listed here. Some, such as some special page table modes, are used to make porting easier.

Because any data at the location specified for the hypercall page will be overwritten by the hypervisor, it is important that there be nothing important there in the guest's address space already. One way of doing this is to specify an address near the end of your address space, and make sure that you keep track of where it is.

An easier way is to simply instruct our assembler to leave a blank page in the middle of our kernel image. We can do this using the .org command, which advances the location counter to an absolute position in the current segment. Listing 2.4 shows how to leave two single page gaps, at pages one and two. If we export these symbol names as globals, we can reference them from our C code later on. Note that neither of these pages is really needed for our trivial kernel, but it is worth defining them now so we don't need to modify our trampoline code later.

[4] In some situations it is possible for a guest to request extra memory as it runs. This is covered in a later chapter.

Listing 2.4: Leaving gaps for the shared info and hypercall pages [from: examples/chapter2/bootstrap.x86_32.S]

```
28          .org 0x1000
29  shared_info:
30          .org 0x2000
31
32  hypercall_page:
33          .org 0x3000
```

2.8.1 The Guest Entry Point

After the Xen loader has parsed this header and mapped everything into memory at the correct locations, it passes control over to the guest kernel. As with a userspace program, the location in which this happens is determined by a symbol name. In the case of a Xen guest kernel, the entry point used by the generic loader is `_start`.

Listing 2.5 shows a simple trampoline for calling a C function `start_kernel`. It is possible to avoid using such a trampoline and simply define a C function called `_start()`. This is not advised, however, because you will need some architecture-specific code in this function, which is better separated out into a trampoline.

Listing 2.5: A simple launch trampoline [from: examples/chapter2/bootstrap.x86_32.S]

```
17  _start:
18          cld
19          lss stack_start,%esp
20          push %esi
21          call start_kernel
```

This trampoline does very little. It begins by clearing the direction flag, then sets up the stack, and calls the real start function. The value of **ESI** is pushed on to the stack so that the start function receives it as the first argument. The value in **ESI** is the address of the start info page, discussed in the next chapter. On x86-64, this trampoline looks different in a couple of ways; the operations are all replaced with their quad-word counterparts and the parameter is passed in a register rather than on the stack. After the trampoline has called the `start_kernel()` function, however, the C code describing the kernel can be the same on both platforms.

Our simple kernel could almost run now. Of course, it would start and then immediately terminate without telling Xen that it had done so, which could cause some problems. One thing is still missing, however. There is currently no way of receiving Xen events. A guest kernel is expected to set up handlers to receive events at boot time. Without these, there is no real way for this simple kernel to

respond to the outside world. For now, we will ignore them, and revisit them in a later chapter. All that remains is to define the C entry point.

Listing 2.6: The simplest kernel [from: examples/chapter2/kernel.c]

```
1 #include <stdint.h>
2 #include <xen.h>
3 #include "debug.h"
4
5 /* Some static space for the stack */
6 char stack[8192];
7
8 /* Main kernel entry point, called by trampoline */
9 void start_kernel(start_info_t * start_info)
10 {
11     HYPERVISOR_console_io(CONSOLEIO_write,12,"Hello World\n");
12     while(1);
13 }
```

Listing 2.6 defines our trivial kernel. We are including the public Xen header, because we need the definition of the start_info_t structure. We are also including `stdint.h`, because this defines a number of fixed-size integer types, which are required by the Xen header.

The kernel stack is defined here as well, although it was referenced elsewhere. For now, we are using a simple, static, 8KB stack for the kernel. This is excessive at the moment, because the trivial kernel only ever uses a single stack frame and doesn't do anything with it. It does give us a little room to play with later, however.

After we are in our kernel, we just infinite loop. This is not particularly exciting, but it does allow us to check that the kernel is running. We could, alternatively, simply exit, but this would not allow us to easily determine the difference between the kernel working and exiting immediately, or crashing and exiting immediately.

If the hypervisor has been compiled with debugging support, then you have one extra facility available to you: the debugging console. This is accessed via the HYPERVISOR_console_io hypercall that we define in `debug.h` (see Listing 2.7). This is a simple hypercall, which writes the contents of the given string to the console.

Listing 2.7: Debugging console support header [from: examples/chapter2/debug.h]

```
1 #include <stdint.h>
2 #include <xen.h>
3
```

```
4 #define __STR(x) #x
5 #define STR(x) __STR(x)
6
7 #define _hypercall3(type, name, a1, a2, a3)          \
8 ({                                                   \
9     long __res, __ign1, __ign2, __ign3;             \
10    __asm volatile (                                \
11        "call hypercall_page + ("STR(__HYPERVISOR_##name)" * 32)"\
12        : "=a" (__res), "=b" (__ign1), "=c" (__ign2),    \
13        "=d" (__ign3)                               \
14        : "1" ((long)(a1)), "2" ((long)(a2)),       \
15        "3" ((long)(a3))                            \
16        : "memory" );                               \
17    (type)__res;                                    \
18 })
19
20 static inline int
21 HYPERVISOR_console_io(
22     int cmd, int count, char *str)
23 {
24     return _hypercall3(int, console_io, cmd, count, str);
25 }
```

If the hypervisor was not compiled with debugging support, this hypercall fails. This does not matter, particularly to our simple kernel. It outputs "Hello World." to the emergency console if debugging support is available, and then continues to infinite loop whether it failed or not. This does, however, give an example of the use of a hypercall. In this case, the hypercall takes three arguments: the command (write, in this case), the length of the string, and the string itself. These are stored in **EBX**, **ECX**, and **EDX**, respectively. In the case of the string, a pointer to the start is stored in the register. Note that this pointer is relative to the guest's address space; the hypervisor's first task is to translate it into a machine address. The prototype for the function inside the hypervisor is:

```
long do_console_io(int cmd, int count, XEN_GUEST_HANDLE(char)
    buffer);
```

Note the type of the third argument. This will be described in more detail later, but for now remember that this macro is used to indicate pointers in the guest's address space. The definition of this macro is architecture-specific, and provides a mechanism for accessing data from a pointer supplied by the guest.

2.8.2 Putting It All Together

Testing our simple kernel requires a few steps. First, we must compile the sources and produce a kernel binary, then we need to tell Xen how to launch it. For now, it doesn't understand any devices, so the last part is pretty easy.

The example code assumes that you have the Xen sources in a `xen` subdirectory of wherever you put the examples. The best way of doing this is to download the latest sources, untar them, and then link them into your examples directory. Later examples will assume Xen has been installed and that the headers are in their standard locations.

Listing 2.8 shows how to build the simple kernel. The first line shows the flag passed to the C preprocessor to tell it where to find the Xen headers. If you didn't link Xen as suggested earlier, you can change this to point to your install location.

Listing 2.8: Makefile for the simple kernel [from: examples/chapter2/Makefile]

```
1  CPPFLAGS += -I../xen/xen/include/public
2  LDFLAGS  += -nostdlib -T example.lds
3  CFLAGS   += -std=c99
4  ASFLAGS   = -D__ASSEMBLY__
5
6  .PHONY: all
7
8  all: testkernel
9
10 testkernel: bootstrap.x86_32.o kernel.o
11     $(CC) $(LDFLAGS) $^ -o testkernel
12
13 clean:
14     rm -f *.o
15     rm -f testkernel
```

The last two flags are going to be used for C and assembly code, respectively. For C code, we need to specify the C99 dialect, because GCC still defaults to C89. For assembly, we need to define the __ASSEMBLY__ macro. This is used in a number of Xen headers. Any assembly file ending in `.S` (as opposed to `.s`) is passed to the C preprocessor before being handed to the assembler. This allows the assembly code to use the macro facilities of the C preprocessor, but doesn't allow it to understand C function prototypes, and so on. Many of the Xen headers are designed to be included in both C and assembly files, and so use the presence of the __ASSEMBLY__ macro to indicate that anything that is specific to C should be omitted.

The linker flags are the most interesting. Because you are building a kernel, the linker is expected to work in a slightly different way than when creating a normal program. The first flag tells it not to include the standard C library. Although it

might be useful to have some things defined, a lot of C library functions rely on system calls, and so won't work inside the kernel. If you accidentally used one of these, you would jump into your own interrupt handler[5] and generate some quite strange results. It is better to copy the functions you actually do want into the kernel tree, and make sure that they don't depend on any external behavior. The other flag tells the linker to use the specified script.

The linker script is shown in Listing 2.9. A linker script is a basic recipe for defining how the various parts from the supplied object files are assembled. We begin with a few simple definitions, setting the output format to ELF and the architecture to i386. We also define the program entry point as _start. Next, we define the structure of the rest of the file. This is fairly standard, with text, read-only data and data sections being placed in order.

Listing 2.9: Linker script for the simple kernel [from: examples/chapter2/example.lds]

```
1  OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
2  OUTPUT_ARCH(i386)
3  ENTRY(_start)
4  SECTIONS
5  {
6    . = 0x0;                  /* Start of the output file */
7
8    _text = .;                /* Text and read-only data */
9
10   .text : {
11     *(.text)
12   } = 0x9090
13
14   _etext = .;               /* End of text section */
15
16   .rodata : {               /* Read only data section */
17     *(.rodata)
18     *(.rodata.*)
19   }
20
21   .data : {                 /* Data */
22     *(.data)
23   }
24
25   _edata = .;               /* End of data section */
26
27 }
```

[5]The trivial kernel doesn't have an interrupt handler yet, but you will add one in a later chapter.

Returning to our Makefile, the rest of the lines simply define how to link the final product and how to clean up afterward. The rest of the build is performed by the implicit make rules. We can now build our kernel:

```
$ make
cc -D__ASSEMBLY__  -I../xen/xen/include/public  \
-c -o bootstrap.x86_32.o bootstrap.x86_32.S
cc -std=c99 -I../xen/xen/include/public  -c \
-o kernel.o kernel.c
cc -nostdlib -T example.lds bootstrap.x86_32.o\
kernel.o -o testkernel
```

Note that the implicit rules that we rely on are present in GNU make, but may not be present in other implementations. Because we are using GNU Compiler Collection extensions to C (for inline assembly), it doesn't seem unreasonable to also use GNU extensions to the UNIX make syntax. If you are using a non-GNU platform, you can either explicitly add the implicit rules or install GNU make. You might find it already installed as `gmake`.

Now that we have built `testkernel`, the next step is to try launching it. Creating a new domain is done using the `xm` command from Domain 0. This takes a configuration file as an argument. We will create a simple one as shown in Listing 2.10. This specifies the name of the kernel, the amount of RAM to allocate, the name to give the domain, and the behavior in case the domain crashes. Actually, 32MB is far more than our trivial kernel will use, but that gives us a little room to expand, and postpones the need to modify the domain configuration file. If the kernel crashes, we destroy the domain; there's not much point doing anything else at this stage.

Listing 2.10: Domain configuration for the simple kernel [from: examples/chapter2/domain_config]

```
1  #  -*- mode: python; -*-
2  #============================================================
3  #Python configuration setup for 'xm create'.  This
4  #script sets the parameters used when a domain is
5  #created using 'xm create'.  You use a separate script
6  #for each domain you want to create, or you can set the
7  #parameters for the domain on the xm command line.
8  #============================================================
9  #Kernel image file.
10 kernel = "testkernel"
11 # Initial memory allocation (in megabytes) for the new
12 # domain.
13 memory = 32
14 # A name for your domain. All domains must have
```

```
15 # different names.
16 name = "Simplest_Kernel"
17
18 on_crash = 'destroy'
```

With the configuration file finished, you can try starting the domain. This is done with `xm create`. After creating the new domain, you can use `xm list` to see that it is still running:

```
# xm create domain_config
Using config file "domain_config".
Started domain Simplest_Kernel

# xm list
Name                ID Mem(MiB) VCPUs State   Time(s)
Domain-0             0       64     1 r-----     37.4
Simplest_Kernel      8       32     1 ------      3.5
```

If you leave the domain running for a while longer, and do `xm list` again, you see the following:

```
# xm list
Name                ID Mem(MiB) VCPUs State   Time(s)
Domain-0             0       64     1 r-----     37.4
Simplest_Kernel      8       32     1 ------    153.7
```

Note that our new kernel has used a significant amount of CPU time in the intervening period. This is not entirely ideal, but it does show that the kernel is working. At the moment, it is consuming as much CPU time as the hypervisor can give it in order to run its idle loop. In future chapters, we will look at how to make it do more interesting things.

Chapter 3

Understanding Shared Info Pages

When an operating system boots, one of the first things it typically does is query the firmware to find out a little about its surroundings. This includes things like the amount of RAM available, what peripherals are connected, and what the current time is.

A kernel booting in a Xen environment, however, does not have access to the real firmware. Instead, there must be another mechanism. Much of the required information is provided by shared memory pages. There are two of these: the first is mapped into the guest's address space by the domain builder at guest boot time; the second must be explicitly mapped by the guest.

The shared info pages do not completely replace a BIOS. One of the main uses for system firmware at boot time is enumerating hardware devices. Whereas the start info page provides a means of mapping a console device, other devices must be found through the XenStore, which provides an interface reminiscent of the OpenFirmware device tree found in most SPARC and PowerPC systems. The console device is made available via the start info page for debugging purposes; debugging output from the kernel should be available as early as possible, and if the kernel is required to interrogate the XenStore before doing this, any bugs that appear while trying to do this will be hard to find.

3.1 Retrieving Boot Time Info

A kernel starting up in Xen begins with a page mapped into its address space containing information about the system, known as the *start info page*. The way

Memory Types

This section will refer to machine and pseudo-physical memory. These terms will be discussed more fully in Chapter 5. The short definition is that machine addresses point to real, physical, memory locations. Pseudo-physical memory addresses point to the guest kernel's virtual address space, and appear as physical memory to parts of the kernel that are not virtualization-aware, and as virtual memory to Xen. The kernel may also implement virtual memory on top of the pseudo-physical memory for userspace applications, giving three kinds of memory address.

the address of this page is transferred to the guest is architecture-dependent; on x86, the address is stored in the **ESI** register.

The contents of this page is defined by a C structure, which is declared in `xen/include/public/xen.h`. Typically, you declare a pointer to a structure of the correct type, and then set it before your kernel does anything else. Listing 3.1 shows how this can be done for x86.

Listing 3.1: Mapping the boot time info page

```
1 start_info_t *start_info_page;
2 asm("")
3     :"=S" (start_info_page));
```

After you have set the pointer correctly, you can interact with the start info page in exactly the same way you would with any other structure. The fields in it provide all of the information required to bootstrap a kernel.

Another option for loading the start info page is to push the value of **ESI** onto the stack and then call the "real" kernel entry point, which then has the start info structure address passed as an argument. This trampoline mechanism is generally cleaner, because it allows you to separate out the architecture-specific code.

Before proceeding, it is a good idea for a kernel to perform some basic validation and check that it is booting in a compatible version of Xen. The magic value in the structure is a string of not more than 32 characters of the form "Xen-*version.sub version*" and can be used for this purpose. In addition to checking that the version of Xen is supported, this ensures that the start info page has been mapped correctly. If it doesn't start with "Xen-", something is seriously wrong, and the best thing to do is abort. Xen guarantees backward compatibility within major versions, although features may be added between minor versions. You should check that the major version is equal and the minor version is not lower than the version on which the kernel was tested.

When you are happy that you are running in a supported Xen environment, booting can proceed. A kernel typically needs to know early on how much RAM

it has available to it, and how many CPUs. The number of pages is provided by the start info structure in the nr_pages element. The number of CPUs, however, is only provided in the shared info structure, discussed in the next section. Typically, a guest will boot on a single (virtual) CPU, and then bring others up after initialization.

Several of the other fields in the structure refer to other pages in memory. These are given in machine addresses; the guest must issue hypercalls to map them in to its own address space. Because they are machine addresses, they are subject to change if the virtual machine is suspended and resumed later, or migrated. For this reason, they should be remapped every time the virtual machine is resumed. The use of machine pages here simplifies the work of the hypervisor and domain builder somewhat, and gives some flexibility to the guest, allowing it to map the pages at convenient locations as required. This comes at the expense of a small amount of extra effort for kernel developers.

The shared_info field is the first to specify a machine page. In this case, it is the one containing the shared info structure discussed in the next section. Mapping this is typically one of the first things a guest should do, because it contains a significant amount of information that can be useful near to system start.

The flags value contains any optional settings for this domain. These are defined in `xen.h` and have the SIF_ prefix. Currently, SIF_PRIVILEGED is defined to indicate a privileged domain and SIF_INITDOMAIN to indicate the initial control domain. The remaining 30 bits of this field are reserved for future use.

The next two fields relate to the XenStore, described in Chapter 8. These form a pair of a form that is used in many places in Xen. The first, store_mfn, gives the machine address of the shared memory page used for communication with the XenStore. The second, store_evtchn , gives an event channel used for notifications.

The console field is a union, and has different values depending on the type of domain. The union is shown in Listing 3.2. Which half of the union is used depends on whether the domain is privileged or not. The Domain 0 guest uses the dom0 part, which contains the memory offset and size of the structure used to define the Xen console. This has one of two modes: text or VGA. For debugging purposes, it is common to run Xen with the console redirected to a serial line, in which case the text console is used. In normal use, the VGA[1] console is available, and can be used for simple graphics display. For more complex graphics, a privileged guest should be given access to the display hardware; it can then run an X server, and display graphics for guests as it would any other networked machines.

For guests in unprivileged domains, the domU part of the union is used. The fields in this represent a shared memory page and event channel used to identify the console device. This is the only device exposed in this way; network interfaces, block devices, and other device types must all be located via the XenStore.

[1]The VGA console is actually an SVGA console, allowing a number of VESA 1.2 modes.

Listing 3.2: Union used to convey console information to guests [from: xen/include/public/xen.h]

```
508     union {
509         struct {
510             xen_pfn_t mfn;       /* MACHINE page number of
                    console page.    */
511             uint32_t  evtchn;    /* Event channel for console
                    page.         */
512         } domU;
513         struct {
514             uint32_t info_off;   /* Offset of console_info
                    struct.          */
515             uint32_t info_size; /* Size of console_info struct
                    from start.*/
516         } dom0;
517     } console;
```

Previous versions of Xen did not use the union, and the fields representing the console were exposed directly as console_mfn and console_eventchn. Macros exist for backward compatibility, as long as the guest is compiled with an old interface version. It is not recommended to use them for new guests, however.

The fields described so far are available at all times, and are updated every time the virtual machine is resumed. The remaining fields are only filled in at initial boot time. The values they represent can be affected by the running of the machine, and so they should not be relied on after boot time.

The next three fields are related to memory management. The pt_base field contains the pseudo-physical address of the page directory. The mfn_list and nr_pt_frames fields, respectively, contain the (pseudo-physical) address of a list of page frames owned by the domains and the number of frames in the list. These values are all computed by the domain builder, which allocates some memory to the domain when it is created.

The next two fields, mod_start and mod_len, relate to module loading. The Xen domain builder can load arbitrary files into the guest kernel's address space. This is useful for bootstrapping, because it allows the kernel to get access to a file before it has loaded block device or filesystem drivers. This can be used for loadable kernel modules, initial ram disks, and so on.

The rest of the start info page is used to pass command-line arguments to the kernel. The cmd_line field contains a string of at most MAX_GUEST_CMDLINE characters, containing any parameters the administrator chose to pass to the kernel.

3.2 The Shared Info Page

The shared info page is used throughout the runtime of a guest kernel to retrieve information about the global state. Unlike the start info page, the shared info page contains information that is dynamically updated as the system runs. Another difference is the way in which the two pages are mapped. The start info page is mapped into the new domain's address space by the domain builder, whereas the shared info page must be explicitly mapped by the guest. The mechanism for doing this is discussed in Chapter 5.

The shared info page is defined by a number of nested C structures, as shown in Figure 3.1. The top level is the shared_info_t . This contains information related to the running *virtual CPU*s, available event channels, wall clock time, and some architecture-specific information.

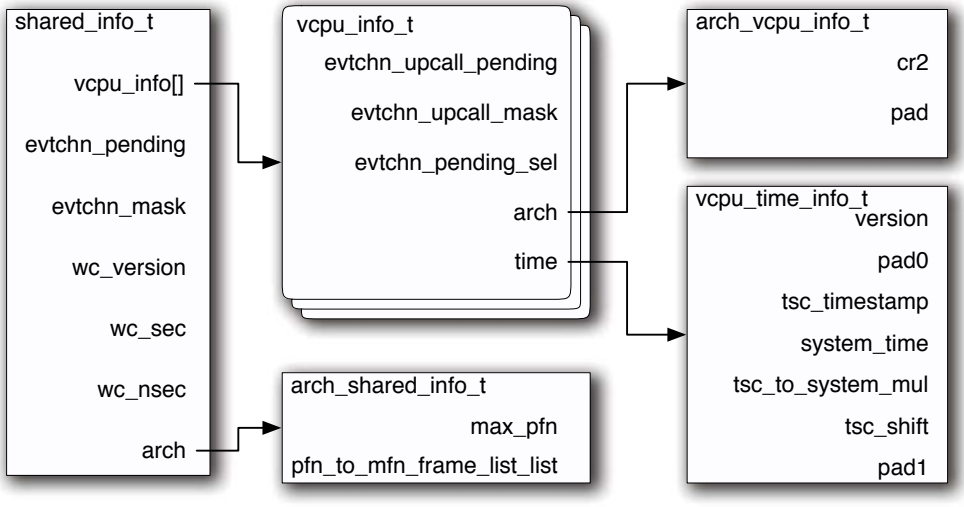

Figure 3.1: The hierarchy of structures used for the shared info page

The first field, vcpu_info, is an array of vcpu_info_t structures. Each of these structures corresponds to a virtual CPU assigned to the domain. If the guest has fewer than MAX_VIRT_CPUS virtual CPUs assigned to it, some of the entries will not be filled in. Trying to bring up nonexistent VCPUs will fail.

Each virtual CPU has three flags relating to virtual interrupts (asynchronously delivered events). The evtchn_upcall_pending field is used by Xen to notify the running system that there are upcalls (interrupts) currently waiting for delivery on this virtual CPU. This is only asserted while events are masked using the

evtchn_upcall_mask flag. This mask prevents any upcalls being delivered to the running virtual CPU. To prevent race conditions, the hypervisor only inspects or modifes these fields from the physical CPU that is running the virtual CPU. The final field in this category is evtchn_pending_sel, which indicates which event is waiting. The event bitmap is an array of machine words, and this value indicates which word in the evtchn_pending field of the parent structure indicates the raised event. For example, if event 12 has been raised, evtchn_pending_sel will be set to 0, because event 12 will be in the first word of the array.

The other two fields in the vcpu_info_t structure contain other structures, one relating to time and the other containing architecture-specific information. On x86, the architecture-specific component contains the virtual CR2 register. This register contains the linear address of the last page fault, but can only be read from ring 0. This is automatically copied by the hypervisor's page fault handler before raising the event with the guest domain. On PowerPC and Itanium platforms, this structure is empty.

Finally, each vcpu_info_t structure contains a vcpu_time_t. This, along with a number of fields sharing the wc_ (wall clock) prefix in the shared info structure, is used to implement time keeping in paravirtualized Xen guests, and will be discussed in detail in the next chapter.

As mentioned earlier, the shared info structure contains a evtchn_pending field. This is eight machine words long, and contains a bit-field indicating which event channels have events waiting. This restricts guests to 256 or 512 event channels on 32- and 64-bit systems, respectively. Bits in this field are set by the hypervisor and cleared by the guest. There is another bit-field in this structure, evtchn_mask, which determines whether an event on a particular channel should be delivered asynchronously. Every time an event is generated, the corresponding bit in evtchn_pending is set to 1. If the corresponding bit in evtchn_mask is set to 0, the hypervisor issues an upcall and delivers the event asynchronously. This allows the guest kernel to switch between interrupt-driven and polling mechanisms on a per-channel basis. In general, polling is likely to be more efficient on very busy channels.

The final field in the shared info page contains architecture-specific information. On x86, this contains two fields, max_pfn and pfn_to_mfn_frame_list_list related to pseudo-physical to machine memory mapping. The first contains the maximum physical frame number and the second the machine address of the frame that contains list of frames containing the pseudo-physical to machine mapping table.

On PowerPC, there is no architecture-specific shared info. The Itanium version of Xen uses this to contain the physical frame number or the start info page and the interrupt vector for the event channel.

3.3 Time Keeping in Xen

Generally speaking, there are two kinds of time that a Xen guest needs to keep track of. The first of these is wall-clock time—the real time that has elapsed. This is predominantly useful for userspace applications that run scheduled tasks, display clocks, and so on. It is also useful for timestamping events, such as filesystem activities. The second is virtual time—the amount of time the guest has spent running.

Virtual time is essential for scheduling of tasks running within a domain. Consider the case of two domains running on the same machine, each of which is scheduled for 10ms at a time. If each domain is running two tasks, and scheduling them for 10ms each using wall time, then one task in each domain will get half of the real CPU's time and the other one will get none.

Within each of these categories, there are two ways in which time can elapse: while the guest is running, and while it is not. While a guest is scheduled, it receives a periodic tick event every 10ms. This allows it to keep track of virtual time easily. Real time is a little more tricky.

Three time values are used to track real time:

Initial system time is time of day when system time is zero. This value is exported in the wc_ prefixed fields in the shared info page.

Current system time is the time that has elapsed since the virtual machine was resumed, and is updated whenever the guest is scheduled.

TSC time is the number of cycles that have elapsed since an arbitrary point in the past. This is named after the *Time-Stamp Counter* (*TSC*), which provides high resolution timekeeping on x86 systems.

The TSC register on a modern x86 chip contains a 64-bit timestamp which is incremented roughly every clock cycle. On very recent chips, it is slightly less frequent; testing on a 2.16GHz Core 2 Duo indicates that it is incremented roughly every four cycles. Two things are guaranteed, however. The first is that it is monotonic—that is, every time you read it, the value will be larger than the last. The second is that the rate at which it is incremented is constant relative to the CPU clock.[2] This allows it to be used for fine-grained correction of timings provided by the hypervisor.

Xen provides a way of translating a TSC difference into nanoseconds with a shift and a multiply. The tsc_shift and tsc_to_system_mul fields in the vcpu_time_info structure provide the shift and multiplication factors, respectively. The function shown in Listing 3.3 translates a TSC change into a number of elapsed nanoseconds.

[2]Note, however, that the CPU clock is not necessarily constant. If the clock is scaled back for power saving, the rate at which the Time Stamp Counter increments may also change.

Do We Need System Time?

It might seem that you don't need system time. If you have the wall clock time at system boot / resume, and the Time Stamp Counter when this happened, you ought to be able to calculate the current system time. In theory, this is the case. In practice, the TSC rate accuracy is limited by the accuracy of the timing circuitry in the system, which is typically prone to a small amount of variable skew. Although it gives a very fine-grained value, it does not give a very accurate one. A system clock using just the TSC value will experience drift. To combat this, the Domain 0 guest is expected to run an NTP client or update its clock from a high-resolution time source. This value is then used to update the system time, reducing drift in the guest.

Listing 3.3: Translating a TSC value to a number of nanoseconds

```
1 uint64_t tscToNanoseconds(uint64_t tsc, struct vcpu_time_info *
      timeinfo)
2 {
3     return (tsc << timeinfo->tsc_shift) * timeinfo->
          tsc_to_system_mul;
4 }
```

Calling this with the value extracted from the TSC register (using the RDTSC instruction) only gives you the number of nanoseconds until some arbitrary point in the past, which is not particularly useful. It can, however, be used to determine how much time has elapsed since the system time stamp was written. Every time this is modified by the hypervisor, the TSC value is also written (to the tsc_timestamp field of the vcpu_time_info_t structure). From this, you can calculate the current system time.

After you have the current system time, you add this to the real (wall) time at system time zero, giving you the current time. The wall clock time at system time zero is stored in the wc_sec and wc_nsec fields of the shared info structure.

3.4 Exercise: Implementing gettimeofday()

Implementing the POSIX gettimeofday() function requires the use of the shared memory page, the Time-Stamp Counter, and some simple calculations. This section will describe the implementation of the function.

The gettimeofday() function takes two arguments, the first of which is a pointer to a timeval structure and the second must be a null pointer. It always returns zero.

System Calls

This example will implement a function, not a system call. If you are writing a simple Xen guest that executes everything in kernel space, this function can be called directly. If you are writing (or porting) a more traditional kernel, this function needs to be called by your system call handler. Installing a system call handler will be discussed in Chapter 7.

For implementing this function, we will assume that the shared info page has already been mapped, and is stored in a global variable. Using this, we will first define a macro that will convert a TSC value to a number of nanoseconds. This is shown in Listing 3.4.

Listing 3.4: A macro for converting a TSC value to a number of nanoseconds [from: examples/chapter3/gettimeofday.c]

```
14 #define NANOSECONDS(tsc) (tsc << shared_info->cpu_info[0].time.
       tsc_shift)\
15     * shared_info->cpu_info[0].time.tsc_to_system_mul
```

This macro uses the TSC scale factors from the first virtual CPU. Because TSC values are meant to be synchronized across CPUs, this ought to work in all cases. The next macro we will define stores the current TSC value in a variable.

TSC is a 64-bit integer. The **RDTSC** instruction reads this and stores the lower 32 bits in **EAX** and the top 32 bits in **EDX**. The TSC() macro defined in Listing 3.5 executes the **RDTSC** instruction and then stores the result in the specified variable.

Listing 3.5: A macro for reading the TSC register [from: examples/chapter3/gettimeofday.c]

```
17 #define RDTSC(x)      asm volatile ("RDTSC":"=A"(tsc))
```

After we have the TSC value, we need to subtract from it the value of the TSC register when the times were updated. To read these, we need to examine the version value of the vcpu_time_info_t structure. If the lowest bit of this is 1, then the timer values are being updated, and so we simply spin until this is not the case. When it is a 0, we keep a copy of the value, attempt to read the wall clock values, and then test the version again. If the version number has not changed, we can proceed; otherwise, we loop.

At first glance, it appears that checking the value of the version is not required. The reason it exists is that a guest may be preempted by the hypervisor and have the time values updated while it is reading them.

Listing 3.6 shows an example implementation of the function. This can be broken down into a few main sections.

Listing 3.6: An example gettimeofday() implementation [from: examples/chapter3/gettimeofday.c]

```
19 int gettimeofday(struct timeval *tp, struct timezone *tzp)
20 {
21     uint64_t tsc;
22     /* Get the time values from the shared info page */
23     uint32_t version, wc_version;
24     uint32_t seconds, nanoseconds, system_time;
25     uint64_t old_tsc;
26     /* Loop until we can read all required values from the same
           update */
27     do
28     {
29         /* Spin if the time value is being updated */
30         do
31         {
32             wc_version = shared_info->wc_version;
33             version = shared_info->cpu_info[0].time.version;
34         } while(
35                 version & 1 == 1
36                 ||
37                 wc_version & 1 == 1);
38         /* Read the values */
39         seconds = shared_info->wc_sec;
40         nanoseconds = shared_info->wc_nsec;
41         system_time = shared_info->cpu_info[0].time.system_time
               ;
42         old_tsc = shared_info->cpu_info[0].time.tsc_timestamp;
43     } while(
44             version != shared_info->cpu_info[0].time.version
45             ||
46             wc_version != shared_info->wc_version
47             );
48     /* Get the current TSC value */
49     RDTSC(tsc);
50     /* Get the number of elapsed cycles */
51     tsc -= old_tsc;
52     /* Update the system time */
53     system_time += NANOSECONDS(tsc);
54     /* Update the nanosecond time */
55     nanoseconds += system_time;
56     /* Move complete seconds to the second counter */
57     seconds += nanoseconds / 1000000000;
58     nanoseconds = nanoseconds % 1000000000;
59     /* Return second and millisecond values */
60     tp->tv_sec = seconds;
61     tp->tv_usec = nanoseconds * 1000;
62     return 0;
63 }
```

- Lines 30-37 keep checking the version counters until they show that neither time is currently being updated. These values are cached.
- Lines 38-42 cache copies of the system and wall clock times where Xen will not touch them.
- Lines 43-47 check that neither value was updated while we were caching them.
- Lines 48-55 update the system time.
- Lines 54-58 update the wall clock time.
- Lines 59-62 return the final value.

It is worth noting that the TSC value should only be used in this context. Some Xen events, particularly live migration, can cause the TSC value to run backward briefly. If a new computer is booted, and a long-running domain is migrated to it, the domain will see the TSC value drop after the migration. The hypervisor always updates the wall clock and system time values after a migration, but any other uses of the Time Stamp Counter will likely cause some unpredictable results.

This page intentionally left blank

Chapter 4

Using Grant Tables

Virtual machines do not exist in isolation. They must interact with others on the system for a number of reasons, not least of which is access to hardware. This chapter will examine the grant table mechanism, which allows memory pages to be transferred or shared between virtual machines.

This chapter will examine how to offer a page to another domain, and how to map an offered page. It will discuss briefly how split device drivers, discussed in more detail in Part II, use shared pages to communicate.

4.1 Sharing Memory

Shared memory is the easiest mechanism for *interprocess communication* (IPC) to implement on a single machine. Because each process's address space is simply a subset of a common memory pool, memory can be shared simply by creating overlapping address spaces.

Due to this simplicity, it is a good choice for Xen. The hypervisor provides a mechanism for sharing memory pages, and cooperating domains can implement a policy built on top of this. This mechanism can be used to provide *interdomain communication*, a way for guests to communicate analogous to IPC between userspace processes.

Xen has a lot in common with several microkernel operating systems. Most microkernels, however, implement message passing rather than shared memory as their basic IPC mechanism. This is because message passing is a slightly higher-level abstraction, and can easily be made network-transparent for systems that do not have a common physical memory architecture. Xen, however, is only interested in providing local interdomain communication. If network communication is required, guest kernels can use their existing networking code.

Most UNIX-like systems implement shared memory between processes using the shm_* family of functions. Creating and sharing a segment of memory in a UNIX-like system might be done as shown in Listing 4.1.

Listing 4.1: Creating a POSIX shared memory region

```
1  /* The POSIX way */
2  int fd = open("/tmp/shmfile",
3      O_CREAT | O_TRUNC | O_RDWR,
4      0666);
5  ftruncate(fd, 4096);
6  void * posix_shm = mmap(
7      NULL,
8      4096,
9      PROT_READ | PROT_WRITE,
10     MAP_SHARED,
11     fd,
12     0);
13 /* The System V way */
14 key_t key = ftok("/tmp/shmfile", 0);
15 int sysv_id = shmget(key,
16     4096,
17     IPC_CREAT);
18 void * sysv_shm = shmat(sysv_id, 0, 0);
```

This listing shows two ways of mapping a shared memory region. The older mechanism, introduced with System V, associates each shared memory region with a key (some kind of integer type). Regions of memory can be associated with these keys, then mapped by other processes.

The POSIX mechanism adopts the UNIX philosophy that everything is a file. Shared memory regions are backed by files. The pseudo-file `/dev/null` can be used if no file is available, to explicitly create an anonymous shared memory region.

The biggest thing to note about these is what is hidden from the user. The first is the granularity. Shared memory can typically only be implemented at page granularity. If you try to map something that is not a multiple of a machine page, the kernel will silently extend the region for you (although space off the end of the explicitly requested region may not work consistently). The second thing abstracted from the user is the placing of memory in the process's address space. Although it is possible to explicitly state the location at which the memory will be mapped, it is not required. Finally, there is the backing store. Shared memory on a UNIX-like system is backed by either a real filesystem object or some swap space.

The Xen mechanism is much lower level. Memory can only be shared at a

page granularity. As with System V shared memory, shared pages are identified by an integer, known as a *grant reference*.

The interface to Xen's shared memory mechanisms is via the grant_table_op hypercall. This takes three arguments: the type of operation to perform, an array of structures containing the operations, and the number of operations in the array. The structures used vary depending on the operation performed. They are all defined in `xen/include/public/grant_table.h`. The prototype of the function that wraps this hypercall is shown below:

```
HYPERVISOR_grant_table_op(unsigned int cmd, void *uop, unsigned
    int count);
```

Note that the operation array, passed as the second argument, is a **void***. This is because the hypercall is polymorphic; the type of this argument depends on the value of the first argument.

All operations relating to modifying grant table permissions are performed by directly modifying the grant table. The structure of the table is described later in this chapter.

4.1.1 Mapping a Page Frame

Two operations can be performed via the grant table: mapping or transferring pages. These two operations are conceptually similar; they both involve inserting the physical page(s) to or from the caller's address space. The difference is that mapping leaves the page in the original domain's address space as well, whereas transferring removes the original reference. Mapping is used to create shared memory, whereas transferring is used to move data from one domain to another. Transfers are also used by the balloon driver, to allow the guest in domain 0 to give pages to other domains. Mapping is a "pull" operation: pages are offered and then mapped by the receiving domain. Transferring is a "push" operation: the calling domain sends the page to another.

The GNTTABOP_map_grant_ref command is used to map page frames associated with a given {grant reference, domain} pair. The structure used to encode the operation is shown in Listing 4.2.

The first field in this structure, host_addr, defines the address in the calling domain's address space (that is, the pseudo-physical address) into which the page should be mapped. In this instance, "host" refers to the host CPU. If the page is going to be used for device I/O, it is possible that two addresses are required. Any platform on which Xen runs will have a memory management unit (MMU). This translates addresses referenced in the CPU into a physical address. A device driver typically needs to know the address of the physical address of pages the device might write to. This is further complicated by the fact that modern machines may come with an IOMMU, so the CPU and device both have different addresses

Listing 4.2: Grant table mapping operation control structure [from: xen/include/public/grant_table.h]

```
1  struct gnttab_map_grant_ref {
2      /* IN parameters. */
3      uint64_t host_addr;
4      uint32_t flags;                 /* GNTMAP_* */
5      grant_ref_t ref;
6      domid_t  dom;
7      /* OUT parameters. */
8      int16_t  status;                /* GNTST_* */
9      grant_handle_t handle;
10     uint64_t dev_bus_addr;
11 };
12 typedef struct gnttab_map_grant_ref gnttab_map_grant_ref_t;
```

for a memory page, and neither corresponds to the physical address. If a memory page is being used for device I/O, the dev_bus_addr field will contain the address that can be used by the device to reference this page.

The flags field represents a number of symbolically defined flags that control how the region is mapped. The following flags can be or'd together to provide a final value:

GNTMAP_device_map indicates that the page should be accessible to I/O devices. On systems with an IOMMU, this adds the page into the I/O address space. The dev_bus_addr value in the structure will be filled in on return, if this flag is set.

GNTMAP_host_map maps the page into the caller domain's address space.

GNTMAP_application_map is only valid if GNTMAP_host_map is set. If this flag is also set, the page is mapped with permissions allowing userspace applications to access it; otherwise, it is only accessible to the calling guest's kernel.

GNTMAP_readonly should be set if the page is mapped as read-only. This could be used for unidirectional communication; the originating domain can write to the page, whereas the mapping domain cannot.

GNTMAP_contains_pte is used to indicate the format of the host_addr field. By default, this is as described earlier in this section. Asserting this flag indicates that it is the machine address of the page table entry to update.

Table 4.1: Grant table status codes

Error Code	Meaning
GNTST_okay	Normal return.
GNTST_general_error	General undefined error.
GNTST_bad_domain	Unrecognized domain id.
GNTST_bad_gntref	Unrecognized or inappropriate gntref.
GNTST_bad_handle	Unrecognized or inappropriate handle.
GNTST_bad_virt_addr	Inappropriate virtual address to map.
GNTST_bad_dev_addr	Inappropriate device address to unmap.
GNTST_no_device_space	Out of space in I/O MMU.
GNTST_permission_denied	Not enough privilege for operation.
GNTST_bad_page	Specified page was invalid for op.
GNTST_bad_copy_arg	copy arguments cross page boundary

The last two input fields, ref and dom, identify the correct grant reference and the domain offering it. This pair uniquely identifies an offered page on a single machine.

The remainder of fields in this structure are for output. The status field is the equivalent of a C library function's return value, or a COM HRESULT, used to identify error conditions. Hopefully, this value will be GNST_okay, indicating success. Other possible results are shown in Table 4.1. All error results are guaranteed to be negative, so testing whether an error has occurred can be done simply by checking the sign of the status.

The handle field gives a unique reference for this grant. The hypervisor keeps track of all granted pages. Unmapping the grant must be done explicitly, using this handle.

4.1.2 Transferring Data between Domains

Transferring a frame is done in a similar way to mapping. In this case, the GNTTABOP_transfer command is used. This uses the control structure described in Listing 4.3.

Before a page can be transferred, the receiver must advertise its interest in receiving a transfer. This is done by creating a grant table entry, set up to allow a transfer. One common use for the transfer mechanism is the balloon driver, which permits a domain to expand its memory usage. A guest creates a transfer request in its grant table, then signals the Domain 0 guest. The Domain 0 guest then transfers new pages to the running guest, if there are some available.

The three input parameters of this operation are the domain and grant refer-

Listing 4.3: Grant table transfer operation control structure [from: xen/include/public/grant_table.h]

```
1 struct gnttab_transfer {
2     /* IN parameters. */
3     xen_pfn_t      mfn;
4     domid_t        domid;
5     grant_ref_t    ref;
6     /* OUT parameters. */
7     int16_t        status;
8 };
9 typedef struct gnttab_transfer gnttab_transfer_t;
```

ence of the receiving domain, and the machine frame of the page to be transferred. Transferring is a good way of moving large amounts of data from one guest to another. The cost is fairly close to constant, because it only updates the page tables. This constant cost, however, is relatively high. For small transfers, copying can be much more efficient.

Copying is a much lower constant-cost operation, but the cost scales linearly in terms of data size. Copying between domains requires one domain to be able to access both the source and destination pages at once. This is not a problem if the pages are pre-shared; but if they are not, it removes the advantage over performing a transfer, because you get the cost of the TLB update as well as the cost of copying.

There is one exception to this: The hypervisor already has all of the pages in physical memory mapped into its address space, so it can perform the transfer at a relatively low cost. The control structure for this operation is shown in Listing 4.4.

Listing 4.4: Control structure for a copy operation [from: xen/include/public/grant_table.h]

```
1  typedef struct gnttab_copy {
2      /* IN parameters. */
3      struct {
4          union {
5              grant_ref_t ref;
6              xen_pfn_t   gmfn;
7          } u;
8          domid_t  domid;
9          uint16_t offset;
10     } source, dest;
11     uint16_t      len;
12     uint16_t      flags;          /* GNTCOPY_* */
13     /* OUT parameters. */
```

```
14      int16_t        status;
15  } gnttab_copy_t;
```

A copy operation is semantically very simple. You have an origin, a destination, and an amount to copy. The C function memcpy performs the same operation for a single process. The hypervisor equivalent has a few more things to configure, because the source and destination can be in different domains. The source and dest parameters of this are used to specify the domain, memory page, and offset within this page of both the source and destination.

The source and destination pages are specified by a union. They can be grant references or machine frame numbers. The only limitation is that both must be accessible by the calling domain. In theory, you could use this mechanism to copy data within the calling domain, although it wouldn't be very efficient. By default, the hypervisor assumes that they are machine page frames (from the calling domain's perspective). To indicate they are not, the GNTCOPY_source_gref or GNTCOPY_dest_gref flag must be specified.

Note that it is also possible to use this mechanism to copy data between two domains when neither is the caller. For example, a domain responsible for a network card could be run in a driver domain to prevent bugs in the network driver affecting the stability of Domain 0. The dom0 guest could then initiate copy operations between the driver domain and another domU guest. The unprivileged guest would be completely unaware that a domain other than dom0 was responsible for the network card.

4.2 Device I/O Rings

One of the main uses for shared memory pages is to implement *I/O rings* for communicating between parts of paravirtualized device drivers. These provide a simple message-passing abstraction built on top of the shared memory mechanism provided by Xen.

The I/O rings provide a method for asynchronous communication between domains. One domain places a request in the ring, whereas the other removes it and inserts a response. Because requests and responses are produced at (roughly) the same rate, a single ring can accommodate them both.

The macros for defining rings are in `xen/public/io/ring.h`. These relate to the creation and use of ring buffers. To define a new ring buffer, you need the structures used for requests and responses, as well as a name for the new ring structure.

Each I/O ring has five main components: start and end pointers for the producer and consumer, and the buffer itself. All ring buffers are power-of-two sized. This is an efficiency optimization; it means that the pointers can simply be in-

cremented, and then masked to give locations within the ring. As long as the producer and consumer segments of the ring don't overlap, it will be fine.

Testing for overlap relies on two features of the ring system. The first is the fact that the response part of the ring always starts at the end of the request part. This means that the only possible overlap is caused by the request segment catching up with the response section. This can be tested by simply checking the distance between the request producer and response consumer indexes. If they are the size of the ring apart, the ring is full. Testing for space in the response part of the ring is easier, because a request taken out of the ring can have a corresponding request inserted into it.

Figure 4.1 shows the structure of an I/O ring, where the buffers advance clockwise. When a request is made, the response end pointer is checked to see if there is enough space ahead of the request start. If there is, the request is written into the ring and the request start pointer incremented. The other end reads the requests from the tail, incrementing the pointer at the end. After processing the request, it writes the result to the front of the response section and increments the response start pointer. The caller then removes the response from the buffer and increments the response end pointer. Note that entries in the ring are of a fixed size, defined by the maximum of the largest request or response that can go in the structure. In implementation, this is handled by making the ring an array of unions of the structures used for requests and responses.

I/O rings that are used to transfer a lot of data might be polled on both ends. Those that are used more infrequently use the event mechanism to signal that data is available.

This structure is used often in systems running on top of Xen. Although the hypervisor only provides the mechanism of shared memory, the message passing system is defined in a general way as part of the Xen interface, allowing the same structure to be recycled in many places.

4.3 Granting and Revoking Permissions

Granting and revoking permissions is done via direct manipulation of the grant table. The grant table itself is an array of structures as shown in Listing 4.5.

The last two fields in this are quite simple; they identify the domain to which the rights are granted, and the page frame the entry refers to. The domain (`domid`) is always filled in by the domain creating the entry. If the domain is granting access to one of its own frames, the relevant frame must be identified by the guest. If the entry relates to a transfer, the hypervisor will fill in the frame number after the transfer.

The `flags` field indicates which rights are granted. This is split into two parts.

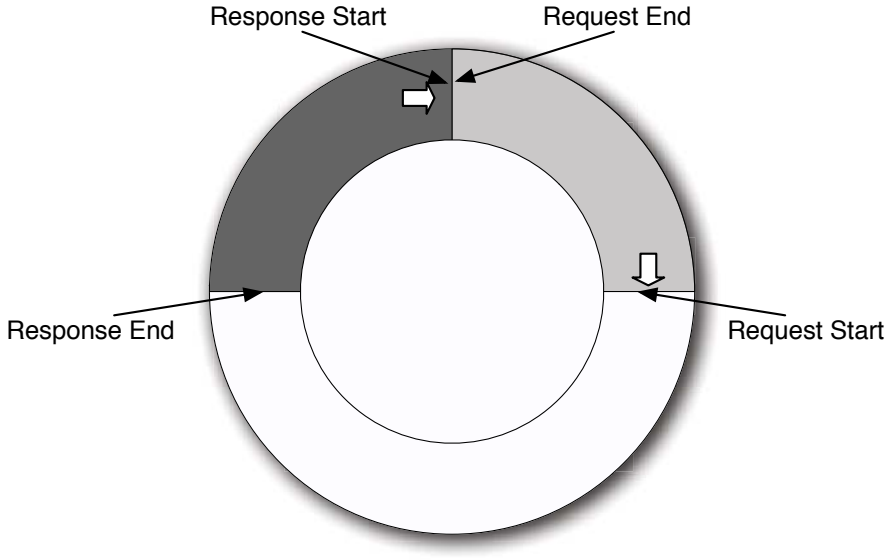

Figure 4.1: The structure of an I/O ring

The first defines the type of the permission granted, whereas the second defines some additional properties of the permission.

Two types of permission are currently supported. These are roughly analogous to read and write permissions on a filesystem. The GTF_permit_access ("read") flag grants the right to the specified domain to map the relevant page into its address space, whereas the GTF_accept_transfer ("write") flag allows the specified domain to transfer a page to this domain.

The type flag can be extracted from the flags field by anding it with the GTF_type_mask constant. Applying this mask to the field returns one of the values discussed previously, or GTF_invalid for an invalid reference.

After the type of a grant has been determined, the remaining bits (the "subflag") of the flags field can be interpreted. For grants permitting access, only one subflag should be set (or cleared) by the guest. The GTF_readonly value indicates whether the receiving guest should be allowed to create read-write mappings. If this bit is set, only read-only maps are permitted. After the page has been mapped, the hypervisor fills in the mode used. The GTF_reading and GTF_writing flags are set if the receiving domain has reading and writing modes, respectively.

For transfer mappings, there are two subflags, both of which are written by the hypervisor. The first, GTF_transfer_committed, indicates that a transfer has

Listing 4.5: Grant table entry structure [from: xen/include/public/grant_table.h]

```
1  struct grant_entry {
2      /* GTF_xxx: various type and flag information.  [XEN,GST]
           */
3      uint16_t flags;
4      /* The domain being granted foreign privileges. [GST] */
5      domid_t  domid;
6      /*
7       * GTF_permit_access: Frame that @domid is allowed to map
            and access. [GST]
8       * GTF_accept_transfer: Frame whose ownership transferred
            by @domid. [XEN]
9       */
10     uint32_t frame;
11 };
12 typedef struct grant_entry grant_entry_t;
```

begun. After setting this flag, the hypervisor is required to complete the operation. While this subflag is set, the guest may not modify the grant table entry. After the operation has finished, and the page has been transferred, Xen asserts the GTF_transfer_completed subflag. After this, the page will have been transferred to the guest's address space, and can be used as any other memory page. The frame field will have been have been updated to reference the transferred frame.

Before the guest can modify its grant table, however, it needs to acquire access to it. This is done using the GNTTABOP_setup_table command in conjunction with an operation of form shown in Listing 4.6.

Listing 4.6: Grant table setup operation control structure [from: xen/include/public/-grant_table.h]

```
1  struct gnttab_setup_table {
2      /* IN parameters. */
3      domid_t  dom;
4      uint32_t nr_frames;
5      /* OUT parameters. */
6      int16_t  status;              /* GNTST_* */
7      XEN_GUEST_HANDLE(ulong) frame_list;
8  };
9  typedef struct gnttab_setup_table gnttab_setup_table_t;
```

The dom field in this indicates which domain the created grant table permis-

sions relate to. Most domains can only modify their own permissions, and so must use DOMID_SELF for this. Domain 0, however, has some extra privileges, and may set up grants for other domains. In general, this should only be used to map pages as readable—transferring a page to an unsuspecting domain or writing data into its address space is likely to cause problems. Marking a page as readable, however, can be used for debugging purposes. A developer can mark a guest kernel's address space as readable, and inspect it while the guest is running, without requiring any modification of the debugged guest.

The size of the created table is set by the nr_frames field. This specifies the minimum size of the newly created grant table. The hypervisor may create a bigger table; however, this will not be visible to the guest. The table is returned in the space allocated by the caller. The XEN_GUEST_HANDLE macro is used in a few places in the Xen API, and is the companion of DEFINE_XEN_GUEST_HANDLE. On x86, the second macro defines a type as a pointer to the argument, with the name __guest_handle_name, where name is the argument. In this structure, the type of frame_list is ulong*. The macro exists to provide some abstraction for references. On x86, it is relatively easy for the hypervisor to access guest memory given a pointer. On other platforms, a little translation is required. These macros provide the abstraction required for this. The set_xen_guest_handle macro should be used to set these. If it is, it should be relatively easy to port guest code between different Xen platforms.

4.4 Exercise: Mapping a Granted Page

The most common use for the grant table is to move blocks of data between the front and back ends of a device driver. The back end receives requests for block I/O operations and processes them. Each operation contains a reference to grant table entry containing the data to be read or written. For a more detailed understanding of how device drivers work, see Chapter 6. For an example of grant table operations, we will take a look at the NetBSD implementation.

Because the grant table operations are performed so often, NetBSD includes a set of wrappers around them in `xen_shm_machdep.c`. The function of relevance in this file is xen_shm_map, which maps a number of grant table entries from a specific domain to a set of virtual addresses in the kernel's address space. The prototype for this function is:

```
xen_shm_map(int nentries, int domid, grant_ref_t *grefp,
    vaddr_t *vap,
    grant_handle_t *handlep, int flags)
```

The arguments are fairly self-explanatory: the number of entries to map, the domain from which to map them, the grant references and destination addresses,

the grant references created by the map, and some flags. The destination addresses and grant references are output parameters; their values are filled in by the mapping function.

The first step in this function is to allocate the address space to store the granted pages. This is done as follows:

```
new_va_pg = vmem_alloc(xen_shm_arena, nentries,
        VM_INSTANTFIT | VM_NOSLEEP);
```

This allocates space for all of the granted pages in an area reserved for the Xen shared memory. A corresponding call to vmem_free() exists in the unmapping function. If there is no space in this region, the function returns ENOMEM. This allows the caller to try unmapping some of its granted pages to try to free more space. The NetBSD implementation attempts to map all of the granted pages specified by a single call to a contiguous region of memory. If the shared memory area has become fragmented, it is possible for larger allocations to fail.

The final step is to actually perform the mapping. This is a two stage operation, as shown in Listing 4.7. Before this can be done, it is necessary to get a pointer to the allocated address space. The vmem_alloc function returns a page number, and so this must be multiplied by the page size to give a pointer. This is optimized into a shift here.

The first stage is setting up a number of gnttab_map_grant_ref_t structures, one per reference. The domain ID and grant reference are filled in as would be expected. The host_addr field in the grant table operation is set to an offset within the allocated block of memory. Finally, the read only flag is translated from the form used in other parts of the NetBSD kernel to the Xen-native version.

Listing 4.7: Mapping a series of granted pages in NetBSD

```
1 new_va = new_va_pg << PAGE_SHIFT;
2 for (i = 0; i < nentries; i++) {
3     op[i].host_addr = new_va + i * PAGE_SIZE;
4     op[i].dom = domid;
5     op[i].ref = grefp[i];
6     op[i].flags = GNTMAP_host_map |
7         ((flags & XSHM_RO) ? GNTMAP_readonly : 0);
8 }
9 err = HYPERVISOR_grant_table_op(GNTTABOP_map_grant_ref, op,
      nentries);
```

After the control structures have been set up, the mapping hypercall is issued. After this, only error checking is required. The new_va value is returned via the vap pointer argument, and the calling function can access the granted pages by looking at offsets within this memory block.

4.5 Exercise: Sharing Memory between VMs

This section will describe a way of sharing a memory page between two cooperating domains. For now, it will be assumed that the domain number and grant table entry have already been communicated via some out-of-band mechanism. Typically, this will be via the XenStore, which is explained in detail in Chapter 8.

The system for sharing a page is split into two components: one must offer the page, whereas the other maps it. We will first look at the guest offering the page, and then the guest mapping it.

The first requirement for presenting a page for sharing is to create a grant. Listing 4.8 shows how this is done.

Listing 4.8: Offering a page for sharing [from: examples/chapter4/offering.c]

```
1  #include <public/xen.h>
2
3  extern void * shared_page;
4  extern grant_entry_t * grant_table;
5
6  void offer_page()
7  {
8      uint16_t flags;
9      /* Create the grant table */
10     gnttab_setup_table_t setup_op;
11
12     setup_op.dom = DOMID_SELF;
13     setup_op.nr_frames = 1;
14     setup_op.frame_list = grant_table;
15
16     HYPERVISOR_grant_table_op(GNTTABOP_setup_table, &setup_op,
           1);
17
18     /* Offer the grant */
19     grant_table[0].domid = DOMID_FRIEND;
20     grant_table[0].frame = shared_page >> 12;
21     flags = GTF_permit_access & GTF_reading & GTF_writing;
22     grant_table[0].flags = flags;
23 }
```

The first step is to create a grant table. This requires some space to be allocated first to store it. Because we are not assuming any malloc() analog in the kernel, we will allocate the space statically. More complicated kernels might incorporate internal heaps, or slab allocators, for this kind of thing.

This example creates a gnttab_setup_table_t , which requests a grant table with

a single element, stored in the grant_table variable. The hypercall is then issued to request, the hypervisor fill in the grant table.

Next, the grant table entry is filled in. Note that the flags field is filled in last. Until the flags field is filled in, the grant table entry is invalid. If we filled in this field before the others, we would have a grant table entry that would appear to be valid, but with a not-fully-defined value. On some architectures, you require a write barrier before filling in the flags value. On x86, however, writes are guaranteed to complete in the order they are issued, so this is not required. One final thing to note is that the address of the shared page is right-shifted by 12. This is to give the page number to which the pointer corresponds. If the least significant 12 bits are not zero, the pointer is not page-aligned, and so this will give some undefined results. The easiest way of ensuring this is to define our shared page statically in our bootstrap as we did with other fixed-size pages.

The second guest needs to take the grant reference and map it to its own address space. For now, we will assume that the grant reference and remote domain have been obtained out of band. A function to perform the mapping is shown in Listing 4.9.

This function takes three arguments. The first is the domain offering the page, and the second is the grant table entry. The third is the location in the caller's address space into which the page should be mapped. This must be page-aligned. The final argument is used to return the handle to the granted page.

The function sets up a single grant table mapping operation. The GNTMAP_host_map variable is set to indicate that we are only mapping this page for interdomain communication, not for talking to a device driver.

Finally, we check the status to see if the mapping works. Using the standard C semantic of a zero return meaning success, and nonzero indicating error, we return -1 if there is an error. In this case, we don't bother to return a grant handle, because it would be invalid.

If everything worked correctly, the caller receives a zero return value, and can then begin using the shared memory page. After it is finished with, the grant handle can be used to unmap the page.

This mechanism can be useful in a few cases. Domains connected to the same NFS server, for example, could use the mechanism to keep a common cache. Similarly, this could be used to implement a virtual network interface similar to the loopback interface, allowing two domains to communicate without having to go via Domain 0. For desktop guests, this could be used to implement something like the MIT shared memory extension to X11, allowing programs on the guest to transfer large amounts of data to the display very quickly.

Listing 4.9: Sharing an offered page [from: examples/chapter4/mapping.c]

```
1  #include <public/xen.h>
2
3  grant_handle_t map(domid_t friend,
4          unsigned int entry,
5          void * shared_page,
6          grant_handle_t * handle)
7  {
8      /* Set up the mapping operation */
9      gnttab_map_grant_ref_t map_op;
10     map_op.host_addr = shared_page;
11     map_op.flags = GNTMAP_host_map;
12     map_op.ref = entry;
13     map_op.dom = friend;
14     /* Perform the map */
15     HYPERVISOR_grant_table_op(GNTTABOP_map_grant_ref, &op,1);
16     /* Check if it worked */
17     if(map_op.status != GNTST_okay)
18     {
19         return -1;
20     }
21     else
22     {
23         /* Return the handle */
24         *handle = map_op.handle;
25         return 0;
26     }
27 }
```

This page intentionally left blank

Chapter 5

Understanding Xen Memory Management

This chapter will explore how memory is handled by Xen. Memory management is generally very CPU architecture dependent. On x86, the operating system sets up page tables that the CPU's *Memory Management Unit* (*MMU*) walks to handle address translation, whereas something like SPARC handles page tables in software and just expects the operating system to fill and empty the *translation lookaside buffer* (*TLB*) in response to page faults. Although Xen runs on several non-x86 platforms, this chapter will focus on x86, beginning with an overview of how x86 manages memory in the absence of virtualization.

Much of this chapter is specific to paravirtualized guests. Those running in HVM mode appear to be running in a normal x86 system, and so can perform memory management exactly as if they were not virtualized. For performance reasons, a guest that detects that it is running in a Xen HVM domain may wish to detect this and switch to using paravirtualized memory management. This is discussed in Chapter 13.

5.1 Managing Memory with x86

Xen, being a paravirtualization solution, closely models the platform on which it runs. Therefore, to understand Xen memory management, it is worth taking a close look at the underlying platform's memory model. For most users, this platform will be x86. If you are already fully conversant with the x86 memory model, please skip ahead to the next section.

Starting with the 80286, the x86 family developed the concept of a protected mode, giving a segmented virtual address space. The 80386 built on this to add

Table 5.1: Segment descriptors on x86

Register	Name	Used for...
CS	Code Segment	Instruction fetches
SS	Stack Segment	Push and pop instructions
DS	Data Segment	Most data addresses
ES	Data Segment	String instruction addresses
FS	Data Segment	Other data addresses
GS	Data Segment	Other data addresses

paging and a 32-bit address space. This situation of paged segments persisted until the Opteron, which discarded the segmented model and moved to a pure paging approach.

The layout of the segments is defined by descriptor tables, stored in memory and referenced by some special-purpose registers. At any given time, there are two descriptor tables, the local and global descriptor tables (LDT and GDT). The LDT defines a set of segments accessible for the current (userspace) process, whereas the GDT defines segments visible to all processes.

Linear memory addresses for use by instructions are generated from two components. The first is a 16-bit segment selector, which identifies the entry in the descriptor table. This then gives a segment start and limit (which are cached in hidden registers). The start is added to the offset specified by the operand to give a virtual memory address. This is known as the linear address, because it corresponds to an address in the process's flat memory space, and is later translated into a physical address by the page table mechanism.

The segment selector is stored in one of the segment registers. These, and their meanings, are listed in Table 5.1. The first three registers all have specific purposes. The **CS** register points to the segment used to contain executable code. Any program counter address is taken to be in this segment. The **SS** register indicates the segment containing the stack. The **ESP** register holds the stack pointer, and any **PUSH** or **POP** instruction uses this address relative to the stack segment. The **DS** register contains the "data" segment; basically, any other data will be stored in this segment.

The **ES** register is used for translating memory addresses to be used by string manipulation instructions. These use the address specified by the **ESI** register, in the segment specified by the **ES** register as a destination. Because string instructions are very rare, it is common to use **ESI** as a general purpose register, and **ES** along with **FS** and **GS** as general purpose segment registers. Most instructions can have memory operands prefixed with the segment register to use, allowing the last four registers to be used to partition a program's memory neatly. In

practice, however, many operating systems simply create a descriptor table with a single segment starting at address 0 and ending at the 4GB boundary, giving a flat address space. Some use a slightly modified version of this, where the code segment is separate, and marked read-only. Another efficiency optimization used by some platforms is to map the entire kernel address space into all processes' address spaces, in a segment with no permissions. This eliminates the need for a context switch when moving between user and kernel space. Operating systems that use this might use 1GB for the kernel and 3GB for userspace processes. This is called a 1GB/3GB memory split. Kernels that perform a context switch for every system call, and thus have completely separate address spaces, are referred to as using a 4GB/4GB memory split. A pure object-oriented system might define independent data and code segments for each object.

The segment selector is a 16-bit value. The first two bits of this indicate the privilege level of the segment. Code running in a lower-privileged ring might be unable to access a given segment. The next bit indicates whether the entry is in the LDT or GDT. The final 13 bits indicate the offset in the table.

Segment Protections

Each segment has a set of access rights associated with it. These allow segments to have some combination of read, write, and execute permissions associated with them. A code segment is typically marked as read and executable, whereas a data segment might be read and writable. For efficiency, some operating systems map a part of the kernel's address space into all processes as a read-only segment, allowing things like coarse-grained time values to be accessed directly, without the overhead of a system call.

Because segments are quite coarse-grained, so are their permissions. Most of the segment-level permissions are mirrored at the page level, allowing finer-grained control. One thing that was missing until recently was the capability to map a page readable, but not executable. Recent chips, however, support the no-execute (NX) bit on page permissions, making this a non-issue.

Because AMD removed the segmenting mechanism with x86-64, it is unlikely that many future systems will make use of segmentation.

After an address has been resolved using a segment and offset, it maps to a linear address. This can either be a physical address (if paging is disabled) or a virtual address (if paging is enabled). Virtual addresses are then passed through the paging mechanism to give a physical address. Typically, pages are 4KB in size (although 2MB or larger pages are allowed by more recent x86 chips).

The paging system introduces two extra layers of indirection when mapping a linear address to a physical one.[1] The linear address is split into three components:

- The page directory entry
- The page table entry
- The offset within the page

The page directory contains a list of page table addresses. Each page directory is one page, as is each page table, meaning that their addresses only need to be 20 bits long (the lower 12 bits will all be zero). The page directory is stored in the page identified by **CR3**, the *page directory base register*. The first 10 bits of the linear address then give an entry in this table.

The page directory entry gives, among other things, the address of a page containing the relevant page table. The next 10 bits of the linear address indicate the page table entry. This, in turn, gives the physical offset of the page. The remaining 12 bits of the linear address then give the address within the page.

The double indirection in the paging system makes it relatively easy to implement aliasing. A single page table can be pointed to by multiple page directory entries, making multiple linear addresses point to the same physical pages.

5.2 Pseudo-Physical Memory Model

Users of modern operating systems are already familiar with virtualization in the context of memory, because the concept of *virtual memory* has been around for quite a while. Typically, the term virtual memory is used to describe memory that can be paged to disk, whereas *protected memory* is used to describe virtualized memory.

In an operating system with protected memory, each application has its own address space. From the application's perspective, it has access to the entire memory space. A hypervisor needs to do something similar for guest operating systems. This means that applications have two layers of indirection to go through. Figure 5.1 shows these three layers.

The triple indirection model is not necessarily required. It would be possible to have all guest kernels exist in the physical address space and simply prevent them from reading from or writing to those owned by other domains. This was not done for two reasons. The first is that many existing operating systems work on the assumption that they have a continuous, flat, address space. Support for gaps in physical memory is typically implemented to enable faulty modules to be

[1] This is slightly different with Page Address Extensions or Page Size Extensions enabled. For details on these modes, consult the *IA32 Architecture Manual*, Volume 3A.

Frame Number Types

At various places in the Xen hypercall API, variables are identified as an *MFN*, *PFN*, *GMFN*, and *GPFN*. The most general of these is a PFN, or *page frame number*, which just means "some kind of page frame number" and can vary in exact meaning depending on the context.

An MFN is a *machine frame number*, which is the number of a page in the (real) machine's address space. This is used when doing pseudo-physical to machine translation, for example. On some architectures—those with hardware virtualization, or software TLBs—the guest is never aware of the machine addresses of any of its memory pages, and so these are not used.

Linear addresses are accessed via a *guest page frame number* (*GPFN*). These are page frames in the guest's address space. These page addresses are relative to the local page tables, and so change quite quickly.

The hardest to define is the *guest machine frame number* (*GMFN*). This refers to either an MFN or a GPFN, depending on the architecture. These addresses are what the guest perceives to be MFNs. On x86 (without HVM), these are real MFNs. On most other platforms, they are GPFNs in a specific context. Any parts of the API that are architecture-neutral tend to use these instead of real MFNs.

ignored. Because it is an exceptional state, it may well be a less optimized code path. Forcing guests to use sparse address spaces would likely either be slow or require invasive modifications to the kernel, neither of which is ideal.

The second reason for adopting this model relates to the virtual machine's life cycle. Even discounting live migration, it is likely that a virtual machine will not spend all of its lifetime in the same place; it will be suspended and later resumed. After resumption, it is not guaranteed that the same physical memory pages will be available—especially if the guest is resumed on a machine other than the one on which it was suspended—and so physical memory addresses stored by the kernel would need to all be remapped.

Most of the time, the guest kernel does not need to know anything about the machine pages. When it does, it must use the translation table provided by the shared info page. On x86, this is part of the architecture-specific shared info structure. Pseudo-physical to machine mappings are highly architecture-specific, and sometimes not required at all. On PowerPC, for example, the native hardware virtualization support is used throughout, and so the guest kernel is never exposed to the machine address space.

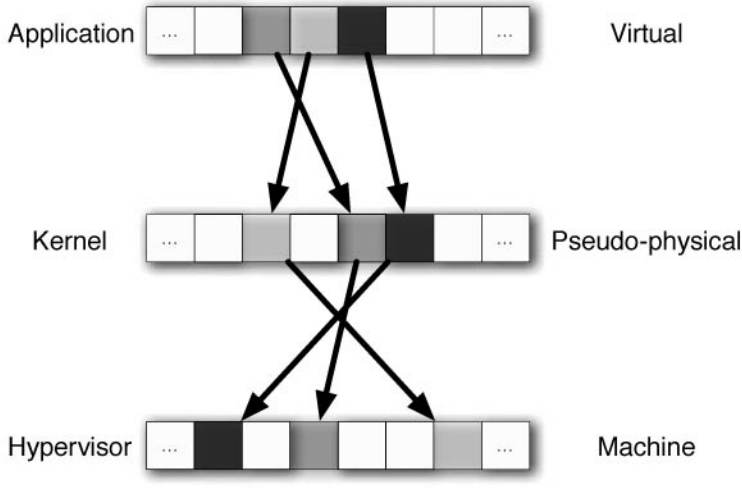

Figure 5.1: The three layers of Xen memory

5.3 Segmenting on 32-bit x86

As described earlier, x86 has a segmented memory model. Although this is not heavily used by most operating systems, it provides some use to Xen. Segments are defined by 64-bit segment descriptors. These have a 32-bit offset, and a 20-bit limit. This leaves 12 bits for flags. One of these defines the granularity of the limit, either page or byte, allowing 0-1MB segments to be defined with a one-byte granularity, or 4KB-4GB segments with a page granularity. The remainder define the type and permissions.

Segments are defined in two tables, the GDT and LDT. Xen guests can update the LDT in the same way they update the page tables, but they can only modify the GDT via an explicit hypercall.

Every time a hypercall is issued, Xen needs to be able to access the hypervisor's memory. Because most hypercalls performed take a pointer as an argument, it also needs to access (some subset of) the guest's address space as well. One way of doing this is to perform a full context switch on every hypercall, and map the required pages from the guest into the hypervisor's address space as required. Although this is conceptually simple, it is fairly slow (around 500 cycles on a Pentium 4), and not the sort of thing you want to be doing very often.

One way around this is to use a memory split, as described earlier. Xen reserves the top 64MB of address space for itself on a 32-bit x86 system. Figure 5.2 shows an example memory layout for a UNIX-like system running under Xen.

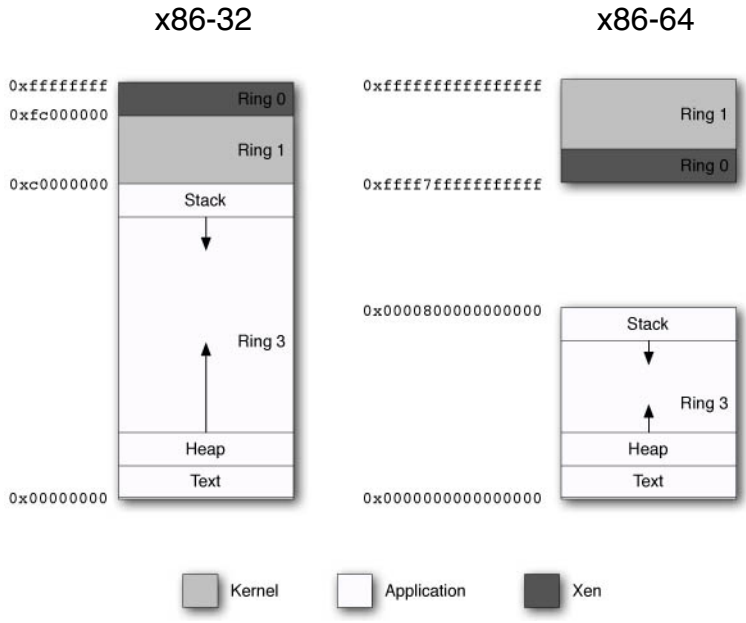

Figure 5.2: Memory layout on x86 systems

The top 64MB are in a segment reserved for Xen. The protection flags for this segment[2] are set to 0, indicating that only ring 0 may access this segment. The remainder of the top 1GB in this example is reserved by the kernel, which is performing the same trick for system calls. Finally, the remainder is used by an application, with the top used for the stack, and the bottom for the heap, growing down and up, respectively. The kernel's segment is marked as only readable by ring 1 or above, preventing the application from touching it, but allowing the kernel and hypervisor access.

When a hypercall is issued, there is a transition to ring 0, and the Xen segment becomes accessible. From there, it is only a matter of referencing the correct segment to be able to switch between Xen, the kernel, and the application's address space.

Although this mechanism is nice and simple, it does have some problems when it comes to portability. Most other modern CPU families don't provide segmentation. In fact, neither does x86-64, meaning this mechanism cannot be used when running Xen in 64-bit mode.

[2]Bits 13 and 14, the descriptor privilege level

The solution is to rely on page-level protections. On x86-64, Xen reserves a large space in the middle of the address space. There is a natural break between 2^{47} and $2^{64} - 2^{47}$ in the current implementation of x86-64. Although the system allows 64-bit pointers, it only actually implements a 48-bit address space[3] with 2^{47} bytes available at the top and the bottom of the address range. Because the x86-64 architecture doesn't support segmentation, the permissions for this must all be done at a page level.

On systems that support the Page Address Extensions (PAE) mode introduced with the Pentium Pro, Xen reserves the top 168MB of a 36-bit address space. In other respects, the memory model used on PAE systems is the same as that on other x86-32 machines.

5.4 Using Xen Memory Assists

The handling of virtual memory is a significant issue for most virtualization systems. In a pure virtualization environment, the hypervisor must trap all page table updates and translate them in such a way that the principle of isolation is not violated. In a paravirtualized environment, hypercalls corresponding to MMU updates are available as a substitute for direct MMU interaction.

This is much faster; however, it is also a lot of work to translate all existing page table updates into hypercalls. For HVM domains, Xen also provides *shadow page tables*. The guest has a copy of the page table, in a set of pages marked by the hypervisor as read-only. When the guest updates these, it causes a fault, and control is transferred to the hypervisor. The hypervisor then translates the updates into the real page table (validating and performing pseudo-physical to machine translation) and continues.

HVM guests require full virtualization of their page tables, so they require shadow page tables, which are quite expensive to maintain and use. Fortunately, some of this extra cost is ameliorated by the fact that VT-x and AMD-V both include hardware support for shadow page tables. The other major use is in migration. When a guest is live-migrated, it is necessary for the hypervisor to track which pages have been modified. The shadow page table mechanism is used in the background during migration to allow this.

Xen also provides an assist somewhere between full paravirtualized page tables and shadow page tables, known as *writable page tables*. In this mode, page tables are not really writable, but the guest is presented with the illusion that they are. The page tables are marked as read-only by the hypervisor, and an attempt to write to them triggers the following sequence of actions:

[3] This is relatively common on 64-bit architectures. A 48-bit address space gives 256TB, which is likely to be enough for the next few years.

1. The page directory entry pointing to the page is invalidated, removing it from the page table.

2. The page is marked as read/write.

3. The guest modifies the page and continues.

4. When an address referenced by the newly invalidated page directory entry is referenced (read or write), a page fault occurs.

5. The hypervisor traps the page fault, validates the contents of the new page table, and reattaches the page.

This is relatively simple to work with, because much existing MMU code can be recycled without major modifications. The major change comes from the fact that the page tables must point to machine addresses, not pseudo-physical ones. This means that the guest kernel's memory management code must deal with machine pages, whereas much of the rest of the kernel will use pseudo-physical addresses. The other modification required comes from the way in which this is implemented. Because the page directory entry is used to cause page faults when referenced pages are touched, the page directory itself can't be modified using this mechanism.

Two other, closely related, assists are available. As mentioned earlier, the segmentation mechanism is used to reserve 64MB at the top of the address space for Xen. This can cause problems for some virtual memory systems that rely on the 4GB segment size.

The two assists related to 4GB segments emulate their existence, and provide a trap when they are used. Interrupt 15, marked as "reserved for future use" in current x86 documentation, is used to deliver the notification. Guests using this assist should install a handler for this interrupt as described in Chapter 7.

All of the assists in this section are enabled or disabled by the hypercall shown here:

```
vm_assist(unsigned int cmd, unsigned int type);
```

The command is either VMASST_CMD_enable or VMASST_CMD_disable, and the type is one of those listed in Table 5.2. The final type has not been discussed here, as it is only applicable in PAE mode.

Table 5.2: Available VM assists

Name	Summary
VMASST_TYPE_4gb_segments	Emulate 4GB segments
VMASST_TYPE_4gb_segments_notify	Raise interrupt (trap) 15 in the guest when the previous assist is used
VMASST_TYPE_writable_pagetables	Make page tables (but not directories) writable directly, not using a hypercall
VMASST_TYPE_pae_extended_cr3	Permit **CR3** to contain a page-directory-pointer-table in PAE mode

5.5 Controlling Memory Usage with the Balloon Driver

Currently, Xen does not do swapping.[4] This means that any memory allocated to a domain is unavailable for other purposes, even if it is not being used. The balloon driver provides a way around this.

By using the balloon driver, a guest can either give up or request more memory from the hypervisor's pool. A well-behaved guest implementing this will return large unused blocks of memory to the hypervisor. A full implementation of the balloon driver will watch a value in the XenStore, which indicates the "target" memory usage. This can be set by a system administrator with access to Domain 0. If the guest is using more memory than requested, it should try freeing some buffers or swapping data out to the block device, and then release some. If it is allowed more, then it may wish to increase the size of its block device cache.

The hypercall used for these operations is HYPERVISOR_memory_op. This takes two arguments: a command and a pointer to a structure containing an operation. A large variety of commands is currently available. Those most relevant to the balloon driver are XENMEM_increase_reservation and XENMEM_decrease_reservation. These both use the control structure shown in Listing 5.1 as the second argument.

Listing 5.1: Memory reservation modification control structure

```
1 struct xen_memory_reservation {
2     XEN_GUEST_HANDLE(xen_pfn_t) extent_start;
3     xen_ulong_t     nr_extents;
4     unsigned int    extent_order;
5     unsigned int    address_bits;
6     domid_t         domid;
7 };
```

[4] This is a design decision. Supporting swapping would mean that guests would not know whether their memory was in-core or not, and could seriously degrade performance.

```
8 typedef struct xen_memory_reservation xen_memory_reservation_t;
```

The first element of this has a variety of different uses. When increasing the reservation, this is filled in by the hypervisor with the machine address of the pages that were allocated to the domain. When decreasing, this is used to pass in the guest machine frame at the base of the range to free.

A third command makes use of this structure, XENMEM_populate_physmap, which is used by the domain builder in Domain 0 to create the initial memory allocation for a new domain. This is generally passed a relatively large allocation of pages in one call.

The next two attributes define the size of the region. The nr_extents gives the number of regions pointed to, and the extent_order gives their size. The order is not an absolute size, but rather the alignment. The actual size is 2^{extent_order}, so quad-word extents would be specified by setting extent_order to 3.

The address_bits field defines the size of the address space of the target. This is used for 32-bit guests on 64-bit systems, for example, and for I/O devices with similar limitations (for example, 32-bit PCI devices). Finally, the domid specifies the domain being referred to, and must be DOMID_SELF for anything other than Domain 0.

Three other commands are relevant to the balloon driver. Both are passed with the second argument set to zero, and return information about the current memory allocation. XENMEM_maximum_ram_page gives the machine frame number of the highest numbered page allocated to the calling guest. On some architectures, the hypervisor does not maintain a machine to pseudo-physical mapping table, and so the guest must do so itself. In this case, the return value from this hypercall can be used to determine how much space the guest needs for this mapping.

The other two relate to the amount of memory allocated to the machine. XENMEM_current_reservation gives the size, in pages, of the current allocation and XENMEM_maximum_reservation gives the maximum. The guest should typically keep track of its own current allocation, rather than frequently query the hypervisor, although this can be useful for validation. The balloon driver, monitoring the target size in the XenStore, should attempt to increase or decrease the reservation until this value corresponds to the target. Any attempts to increase the memory reservation beyond the maximum will fail. Keeping track of the maximum inside the guest can be used to avoid issuing comparatively expensive hypercalls that are known to fail in advance. The maximum RAM page can always be returned; however; the current and maximum reservation size will sometimes fail. When this happens, the hypercalls return negative values.

These commands have quite a bit of flexibility, and are used beyond the balloon driver. In the context of ballooning memory, they are typically used to increase or decrease a guest's memory allocation by one or two pages at a time. They are also quite commonly used to acquire regions of contiguous memory for device use.

5.6 Other Memory Operations

The HYPERVISOR_memory_op hypercall has a large number of commands associated with it beyond those discussed in the last section. Each small group of these has an associated command structure.

The XENMEM_exchange command is used to atomically exchange two page frames in the guest machine page frame space. This could be used, for example, to defragment memory if large contiguous regions are required by the guest. This uses the control structure shown in Listing 5.2. The three elements in this correspond to the input and output addresses, in the form used for the set of commands described in the last section.

Listing 5.2: Page exchange control structure

```
1 struct xen_memory_exchange {
2     struct xen_memory_reservation in;
3     struct xen_memory_reservation out;
4     xen_ulong_t nr_exchanged;
5 };
6 typedef struct xen_memory_exchange xen_memory_exchange_t;
```

The in and out fields of this are filled in with the ranges of page frames to be exchanged, although the address_bits field in in is ignored. When the hypercall is issued, the pages have their location in the guest's machine address space moved from the location specified by in to the location specified by out atomically.

The next command is only relevant on some platforms. If the guest is aware of the pseudo-physical to machine mapping then there needs to be some way of keeping track of this mapping. HVM guests on x86 are generally not aware of this mapping, and neither are guests on most other platform, which use hypervisor-support extensions in the underlying architecture to perform this translation. The XENMEM_machphys_mfn_list command is used to acquire this mapping, using the operation structure shown in Listing 5.3.

Listing 5.3: Control structure for acquiring the machine to pseudo-physical mapping table

```
1 struct xen_machphys_mfn_list {
2     unsigned int max_extents;
3     XEN_GUEST_HANDLE(xen_pfn_t) extent_start;
4     unsigned int nr_extents;
5 };
6 typedef struct xen_machphys_mfn_list xen_machphys_mfn_list_t;
```

This structure is basically a wrapper around an array. The extent_start member is a pointer[5] to a block of memory into which the hypervisor should write

[5] Or other handle, depending on the platform.

the machine to physical table. Generally, this mapping is fairly static, because the hypervisor does not support swapping, although it can be modified by the guest. After migration, this table will be invalid and so should be rerequested if it is required. The max_extents field indicates the amount of space in this array, whereas nr_extents is filled in by the hypervisor to convey the number actually used. This table is an array of pseudo-physical frame numbers of the start of 2MB extents, corresponding to 2MB ranges in the pseudo-physical address space. Some gaps may be present in this, represented by null values in the array.

On some architectures, this table is mapped into the guest's address space at system start. The address can be retrieved using the XENMEM_machphys_mapping command. This takes a control structure with three fields as shown in Listing 5.4. The fields in this are quite simple; they store the range of virtual addresses of the machine to pseudo-physical address table, and the largest page it contains. All of these fields are filled in by the hypervisor when the call returns.

Listing 5.4: Operation for retrieving the virtual address of the M2P table [from: xen/include/public/memory.h]

```
172 #define XENMEM_machphys_mapping      12
173 struct xen_machphys_mapping {
174     xen_ulong_t v_start, v_end; /* Start and end virtual
            addresses.   */
175     xen_ulong_t max_mfn;        /* Maximum MFN that can be
            looked up. */
176 };
177 typedef struct xen_machphys_mapping xen_machphys_mapping_t;
```

Guests on some platforms (including HVM x86), as mentioned earlier, are not directly aware of their machine frame addresses. They simply act as if their pseudo-physical addresses are real addresses. This is not always acceptable, however. If one of these guests passes an address to a piece of hardware in order for it to perform a DMA transfer, the device ends up writing over some random part of the machine address space. In an ideal world, the IOMMU would be set up so that the device's address space directly corresponds with its owners. This is not always possible, however, because not all platforms come with an IOMMU as standard.

If a guest needs to perform this translation (that is, it is a pure paravirtualized guest), it can use the XENMEM_translate_gpfn_list command with the operation structure from Listing 5.5. Another use for this hypercall is for Domain 0 to determine the machine address pointed to by a pointer extracted from the guest domain. The domid, as always, tells the hypervisor which domain the operation applies to. Next comes an array of guest page frame numbers and the number of elements in the array. When this hypercall is executed, each one of these has a corresponding value filled in in the final array, indicating the corresponding

machine frame. As usual, the guest is responsible for allocating the space into which the hypervisor will store this data.

If this hypercall is valid, and no error occurs, this gives a non-negative return value.

Listing 5.5: Control structure for translating a list of GPFNs into MFNs

```
1 struct xen_translate_gpfn_list {
2     domid_t domid;
3     xen_ulong_t nr_gpfns;
4     XEN_GUEST_HANDLE(xen_pfn_t) gpfn_list;
5     XEN_GUEST_HANDLE(xen_pfn_t) mfn_list;
6 };
7 typedef struct xen_translate_gpfn_list
      xen_translate_gpfn_list_t;
```

As discussed previously, the hypervisor has to provide replacement functionality for the BIOS to guests, because they are not able to make BIOS calls directly. One such call, interrupt 15h with **AX** set to E820h, is used to get the initial memory layout at boot time. This call supersedes earlier BIOS memory querying operations designed for real mode operation. The hypervisor replaces this with XENMEM_memory_map. This takes an operation field with only two elements: nr_entries and buffer. The first is filled in with the number of entries for which space has been allocated in the buffer, and the second points to the buffer. After being invoked, the first parameter is set to the number of entries in the buffer.

The buffer itself is filled with entries in the same form as that provided by the corresponding BIOS call. These are composed of a 64-bit base address, a 64-bit length, and a 32-bit type. The type is either 1 for memory that the operating system can use or some other value for other kinds of memory. A guest that makes use of this BIOS call should already have code for handling these structures, and should simply replace the BIOS call with this hypercall. New guests are unlikely to need it while in domain U, although moving to Domain 0 will give them an address layout more similar to native x86, and so they may find it useful. A variant of this, XENMEM_machine_memory_map, uses the same control structure but returns the layout of machine memory, rather than pseudo-physical.

These two have a companion command, XENMEM_set_memory_map, which is used by Domain 0 to initialize the memory map of a newly created domain. This is used in conjunction with an operation structure containing two elements. The first is the ubiquitous domid and the second, map, is a structure of the sort used with the previous two commands. When the guest calls XENMEM_memory_map, the value passed in with map by Domain 0 will be returned.

5.7 Updating the Page Tables

Memory pages in Xen have a type corresponding to their use. Descriptor tables (GDT or LDT), page directories, and tables each have their own types. Any page that has one of these types may be mapped as readable by the domain, but not writable. This ensures that the domain can't ever update its own page tables, which in turn ensures that no unprivileged domain can access another's address space without going via the grant table mechanism.

Because a guest is not permitted to modify its page tables directly, a hypercall mechanism needs to be in place that allows it. This hypercall performs the update, after checking that it should be permitted. Any attempt to modify the page tables to point to a machine page not owned by the guest will fail.

Because hypercalls are significantly more expensive than simple memory accesses, a single hypercall can perform multiple page table updates. The prototype for this hypercall is shown here:

```
HYPERVISOR_mmu_update(mmu_update_t * req, int count, int *
    success_count);
```

As usual, the hypercall takes an array of structures, and a count indicating the number of these structures. The third argument is used to return the number of operations that succeeded. If success_count is less than count, at least one of the operations failed. The hypercall does not indicate directly which updates failed. It is up to the caller to walk the page tables to discover this.

The mmu_update_t structure is a simple pair, as shown in Listing 5.6. The ptr field is used to contain the address of the page table entry to be updated, and the val field is used to specify the new value. Because page table entries are 32 bits[6] (and are page aligned), the least significant two bits of the ptr value is always zero. The hypercall interface specifies that these two bits are ignored when calculating the address. Instead, they are used to encode the type of the update.

Listing 5.6: MMU update control structure [from: xen/include/public/xen.h]

```
329 struct mmu_update {
330     uint64_t ptr;       /* Machine address of PTE. */
331     uint64_t val;       /* New contents of PTE.    */
332 };
```

Two update types are permitted, and a single hypercall may use a mixture of them. Either one of the following values can be XOR'd into the ptr value. Because the lowest two bits of this field should be 0, for alignment reasons, they are reserved for flags.

[6] On 32-bit x86. On 64-bit platforms, they are 64 bits and the least significant four bits of the pointer are unused.

MMU_NORMAL_PT_UPDATE performs a standard page table update. ptr points to an entry in a page table or page directory.

MMU_MACHPHYS_UPDATE performs an update to the machine to physical mapping table. When invoked from a domU guest, val must point to a machine page owned by the domain.

While it is more efficient to update a number of page table entries at once, this is not always required. A userspace program in the guest calling malloc(), for example, may only require a single change to the page table to map a new page into its address space. In this case, having to create the control structure provides some overhead for the guest, whereas having to reference guest's memory provides a small amount of overhead for Xen. A variant of this hypercall, shown below, allows a single entry to be updated.

```
HYPERVISOR_update_va_mapping(unsigned long va, uint64_t val,
    unsigned long flags)
```

On x86, the three arguments to this hypercall are passed in **EBX**, **ECX**, and **EDX**, removing a layer of indirection from the system. The first two arguments do not correspond directly to the ptr and val fields in the previous hypercall. This call updates the current page table so that the page pointed to by va will reference the machine page val. The flags argument indicates which, if any, TLBs are flushed as a result of this update.

The flags variable is split into two parts. The lowest two bits indicate the type of the flush; the remaining bits define the scope. Three types of TLB flush are possible:

- No flush at all, indicated by UVMF_NONE.
- A single entry can be flushed, if the UVMF_INVLPG flag is specified.
- Finally, the entire TLB contents can be flushed if UVMF_TLB_FLUSH is given.

The mask used to extract the type is UVMF_FLUSHTYPE_MASK. The remaining bits are used to indicate which TLBs should be flushed. In a uniprocessor system, there is only a single TLB, so this is not particularly important. On others, it may be that a number of threads belonging to a single process are scheduled to run concurrently on different processors. In this case, they will be sharing the same page tables, so an update needs to be propagated. Two flags can simply be OR'd with the value; UVMF_LOCAL or UVMF_ALL indicate either the local TLB or all, respectively. The final option UVMF_MULTI means *some* TLBs should be flushed—exactly which must be specified with a CPU bitmap. The pointer to this bitmap must be OR'd with the flags argument.

A guest running in Domain 0 has access to a four-argument version of this hypercall: HYPERVISOR_update_va_mapping_otherdomain. This allows a privileged domain to perform updates to another domain's page tables. In most settings, the only privileged domain is dom0, however in more recent versions of the hypervisor it is possible to delegate some of Domain 0's power to other domains using the Xen Security Modules framework.

These hypercalls allow most common MMU operations; however, they don't allow everything a guest might need. An extended interface is provided via the hypercall shown here:

```
HYPERVISOR_mmuext_op(struct mmuext_op *op, int count, int *
    success_count, domid_t domid);
```

The structure of this hypercall is very similar to the nonextended version. An array of operations is passed, and the number that succeeded is returned in the success_count argument. The domid parameter allows operation on other domains page and segment descriptor tables to be performed, as long as the calling domain is sufficiently privileged (typically, it is set to DOMID_SELF). This allows Domain 0 to implement the same functionality as the grant tables provide, although this is not recommended.

The control structure for this operation, shown in Listing 5.7, is comparatively complicated. The first argument is the command to be performed. There are currently 15 of these supported by Xen. The first five relate to the pinning mechanism. If a page used by a table is no longer referenced by any others in the page table hierarchy, the hypervisor no longer gives it special attention. If it is then re-added, the hypervisor needs to revalidate its contents. To prevent this, the page can be "pinned," forcing Xen to treat it as a page table entry, even if it is not referenced.

Listing 5.7: Extended MMU operation control structure [from: xen/include/public/xen.h]

```
236 struct mmuext_op {
237     unsigned int cmd;
238     union {
239         /* [UN]PIN_TABLE, NEW_BASEPTR, NEW_USER_BASEPTR */
240         xen_pfn_t      mfn;
241         /* INVLPG_LOCAL, INVLPG_ALL, SET_LDT */
242         unsigned long linear_addr;
243     } arg1;
244     union {
245         /* SET_LDT */
246         unsigned int nr_ents;
247         /* TLB_FLUSH_MULTI, INVLPG_MULTI */
248         XEN_GUEST_HANDLE_00030205(void) vcpumask;
249     } arg2;
250 };
```

The four commands to pin a page are of the form MMUEXT_PIN_L1_TABLE, where L1 indicates the leaf layer of the page tables, and can be any value between L1 and L4. For a 32-bit x86 system, L1 will be the page table, and L2 will be the page directory. A single command, MMUEXT_UNPIN_TABLE, is used for the inverse operation. For both of these operations, the mfn part of the arg1 union is used to specify the machine frame containing the page table (or directory). Not all of these are valid on all host architectures. On 32-bit x86, for example, the page tables are only two layers by default (although a third layer is possible with some PAE modes), whereas on x86-64, all four layers are used.

The next commands relate to installing new page tables. The MMUEXT_NEW_BASEPTR and MMUEXT_NEW_USER_BASEPTR commands install a new page table base (**CR3** value) in the global and userspace contexts, respectively. The latter command is only valid on x86-64 platforms. In both of these cases, the mfn part of the arg1 union is used to specify the machine page of the new page table base.

The next group of six commands relates to the translation lookaside buffers. These roughly correspond to the operations allowed by the nonextended version of this hypercall. The six commands perform two operations in three scopes; they flush either the entire TLB or a single entry in either the local, a specified set of, or all TLBs. Table 5.3 lists them all.

Table 5.3: Extended MMU operation commands

Command	Operation	Scope
MMUEXT_TLB_FLUSH_LOCAL	Flush TLB	Local TLB
MMUEXT_INVLPG_LOCAL	Invalidate entry	Local TLB
MMUEXT_TLB_FLUSH_MULTI	Flush TLB	Specified TLBs
MMUEXT_INVLPG_MULTI	Invalidate entry	Specified TLBs
MMUEXT_TLB_FLUSH_ALL	Flush TLB	All TLBs
MMUEXT_INVLPG_ALL	Invalidate entry	All TLBs

If a single TLB line is being invalidated, the address must be specified in the linear_addr component of the arg1 union. For the ∗MULTI variants, the vcpumask attribute of the arg2 union is used to specify a pointer to a bitmap specifying which TLBs (identified by virtual CPU number) should be flushed.

The next command is somewhat related; it flushes the contents of the cache. If MMUEXT_FLUSH_CACHE is specified, the contents cache is written back to main memory and all cache lines invalidated. This takes no arguments.

The final command, MMUEXT_SET_LDT, is used to specify a custom LDT. The linear_addr component of arg1 specifies the (page aligned) address of the LDT, and the nr_ents element of arg2 specifies the number of entries it contains.

There is one more function related to memory management that can be performed by a guest; updating the segment descriptor tables. Updates to the LDT can be handled via the HYPERVISOR_mmu_update call described earlier, however the GDT requires special handling.

By default, Xen installs a flat GDT, with a single segment mapping the entire linear address space. Some guests may want to install their own GDT, however. This is typically done simply by issuing the **LGDT** instruction, which loads the operand value into the GDTR. As with other MMU updates, it is necessary for the hypervisor to validate the contents of the new GDT to ensure that it doesn't touch any memory not owned by the domain.

```
HYPERVISOR_set_gdt(unsigned long * frame_list, int entries);
```

The prototype for this hypercall is shown above. The arguments give an array of up to 16 machine frames, and the number of segment descriptors they contain. The maximum number of segments allowed in the GDT by x86 is 8192; however, these are not all available to Xen guests. As discussed earlier, the hypervisor itself uses the segmentation system to isolate the guest. The segments defined by Xen begin on page 14 of the GDT, giving 7168 free entries for use by the guest kernel. Of these, four are initially configured as flat code segments and data/stack segments for ring 1 (the paravirtualized kernel) and ring 3 (userspace applications). A guest using a flat address space can use these values directly, without installing its own GDT. They are symbolically defined as macros of the form FLAT_RING1_CS, where RING1 is replaced by RING3 for the userspace version, and CS can be DS or SS for the data and stack segments, respectively.

5.7.1 Creating a New VM Instance

Creating a new virtual machine instance requires two phases: the creation of the domain and the booting of the guest kernel. These two are in some ways related. When preparing the domain, the start info page must be mapped into its address space at the location specified by the guest kernel, for example.

The initial memory configuration of a new domain contains the guest kernel, any specified modules, and the start info page. It must also contain access rights to the shared info page and grant references relating to device drivers. Each driver must also be properly instantiated and have relevant XenStore information filled in.

The HYPERVISOR_memory_op hypercall can, when used from Domain 0, be used to configure the initial allocation of space to a domain, and the hypercalls described in the previous section can be used to configure the initial page tables. Typically, the domain builder in dom0 creates the domain with the correct amount of memory, maps the required pages, and initializes the required structure. Control is then passed to the new domain via Xen.

5.7.2 Handling a Page Fault

When a page fault occurs, the hypervisor generates a trap, and asynchronously invokes the corresponding trap handler (number 14, in the case of x86). This handler must then either fix the problem or deal with the failure in some other way (for example, terminate the running process).

Interrupts

Xen allows a virtual *interrupt descriptor table*IDT to be installed by guests. In Xen terminology, this is known as a *trap table*. The trap table structure, discussed in more detail in Chapter 7, is structurally similar to the host platform. If porting an operating system to Xen, the existing interrupt handlers can typically be used directly.

The first thing to do is detect which address caused the fault. On native x86, this address is stored in the **CR2** register. Under Xen, the CR2 register is copied into the cr2 element of the arch_vcpu_info structure. As with a native page fault handler, the first thing that must be done is to copy this value somewhere safe. This is because the page fault handler itself accesses memory, and so can cause additional faults. If this occurs (typically indicating damaged page tables or an invalid memory reference), the contents of the CR2 register[7] are overwritten.

After the page fault address has been identified, the page tables must be updated so that it is valid. Typically, this is done in one of two ways. If the page fault is caused by lazy allocation, an unused physical page must be recycled into use. If the fault was caused by swapping, the contents must be loaded from disk to either an unused page or to a page that is then evicted. After the new page has been identified, the guest kernel issues the HYPERVISOR_update_va_mapping hypercall to point the virtual address to the new physical page.

5.7.3 Suspend, Resume, and Migration

Most guests will want to support suspending and resuming. These operations allow the domain to be serialized to disk and then restored later. The machine on which the domain is restored might not even be the same one on which it was suspended. It must meet the following requirements to be restored:

- The CPU type must be the same.
- The new host must have enough free RAM.
- The block device (if one is in use) must be available.

[7] Real or virtualized.

It is possible, for example, to have a block device backed by a partition on a USB Flash drive, and suspend the VM to another partition. The drive could then be carried to a new machine and resumed. It is possible that this kind of migration might cause some problems for a guest, because it will need to reinitialize any network interfaces, and will be likely to suddenly change IP addresses. This shouldn't be a problem for operating systems that already work in laptop environments, however.

The first step in a suspend operation is for the guest to receive a suspend request via the XenStore. It must then put itself into a state ready for suspension and then issue a hypercall to actually begin the suspend operation.

CPU Suspension

Although not related to memory, and thus not entirely relevant here, it is worth remembering that Xen only suspends the state of the first virtual CPU. If you are running on multiple CPUs, it is your responsibility to stop the others before suspending. When you resume, you must then reinitialize the virtual CPUs before continuing.

It is likely that any running guest will have references to machine pages stored at various places. After it is suspended and resumed, these will all point to random locations, many of which might not even be owned by the guest and thus will cause faults if they are used. The guest must turn all of these into pseudo-physical addresses if it owns the pages, or some more abstract representation if not.

An example of the pages requiring more abstract storage is the set used for communication with virtual devices. After migration, the pages need to be reconnected by the new back end. Some part of the guest must maintain the information required to locate the new provider for the device in the XenStore and reconnect the page. Other parts of the guest may use a pseudo-physical page reference, as long as they do not attempt to use it until it has been remapped.

When the guest resumes, it must do the inverse operation. First, shared pages must be remapped via the grant tables. Often these are indirect references, so the XenStore page must be reconnected and then this is used to discover the new pages. Finally, any other virtual CPUs must be restarted and execution can continue.

5.8 Exercise: Mapping the Shared Info Page

One of the first things a guest needs to do is map the shared info page into its own address space. The machine address was passed into the domain via the start

info page, in the start_info field. In the kernel bootstrap we wrote previously, we had reserved space for it, so all we need to do now is update our page tables to incorporate it.

To make the preallocated space accessible from C, we need to tell the compiler about it. We do this by declaring the symbol as **extern**. The compiler then treats it as valid and, if all goes well, the linker replaces it with the address in the assembly file. The full definition looks like this:

```
extern shared_info_t shared_info;
```

This is comparatively simple to do. Because we are only updating a single page table entry, we can simply issue the hypercall as shown in Listing 5.8. The only flag required is to invalidate the TLB line referring to the update.

Listing 5.8: Mapping the shared info page into the pre-prepared space

```
1 HYPERVISOR_update_va_mapping(
2     (unsigned long) &shared_info,
3     (unsigned long long) start_info->shared_info,
4     UVMF_INVLPG);
```

After this has completed, the guest should be able to use the shared_info pointer as it would any other C data structure. Several other pages need to be mapped in this way at start-of-day. These include the XenStore and console driver, provided by the start info structure, and any device-related pages found in the XenStore.

This process must be repeated when the domain is resumed, and so it is typically worth creating a function that maps them all in one go, which can be called when the kernel is booted, and again whenever it is resumed.

Part II

Device I/O

This page intentionally left blank

Chapter 6

Understanding Device Drivers

Device drivers are an important part of any operating system—without them, the kernel (and thus the applications) can't communicate with physical hardware attached to the system.

Most full virtualization solutions provide emulated forms of simple devices. The emulated devices are typically chosen to be common hardware, so it is likely that drivers exist already for any given guest. Examples of hardware emulated include simple IDE hard disks and NE2000 network interfaces. This is a reasonable solution in cases where the guest cannot be modified, and is used by Xen in HVM domains where unmodified guests are run.

Paravirtualized guests, however, need to be modified in order to run anyway. As such, the requirement for the virtual environment to use existing drivers disappears. Making guest kernel authors write a lot of code, however, would not be a very good design decision, and so Xen devices must be simple to implement. They should also be fast; if they are not, they have no advantage over emulated devices.

The Xen approach is to provide abstract devices that implement a high-level interface that corresponds to a particular device category. Rather than providing a SCSI device and an IDE device, Xen provides an abstract block device. This supports only two operations: read and write a block. This is implemented in a way that closely corresponds to the POSIX `readv` and `writev` calls, allowing operations to be grouped in a single request (which allows I/O reordering in the Domain 0 kernel or the controller to be used effectively). The network interface is slightly more complicated, but still relatively easy for a guest to implement.

6.1 The Split Driver Model

Supporting the range of hardware available for a commodity PC would be a daunting task for Xen. Fortunately, most of the required hardware is already supported by the guest in Domain 0. If Xen can reuse this support, it gets a large amount of hardware compatibility for free.

In addition, it is fairly common for an operating system to already provide some multiplexing services. The purpose of an operating system (as opposed to running applications directly on the hardware) is to provide an abstraction of the real hardware. One of the features of this abstraction in a modern OS is that applications are, in general, not aware of each other. Two applications can use the same physical disk, network interface, or sound device, without worrying about others. By piggy-backing on this capability, Xen can avoid writing a lot of new and untested code.

This multiplexing capability is quite important. Some devices on high-end systems, particularly mainframes, are virtualization-aware. They can be partitioned in the firmware, and each running operating system can interact with them directly. For consumer-grade hardware, however, this is not common. Most consumer-level devices assume a single user, and require the running operating system to perform any required multiplexing. In a virtualized environment, device access must be multiplexed before it is handed over to the operating system.

As discussed earlier, the hypervisor provides a simple mechanism for communicating between domains: shared memory. This is used by device drivers to establish a connection between the two components. The I/O ring mechanism, described later in this chapter, is typically used for this.

One important thing to note about Xen devices is that they are not really part of Xen. The hypervisor provides the mechanisms for device discovery and moving data between domains; the drives are split across a pair of guest domains. Typically, this pair is Domain 0 and another guest, although it is also possible to use a dedicated driver domain instead of Domain 0. The interface is specified by Xen; however, the actual implementation is left up to the domains.

Figure 6.1 shows the structure of a typical split device driver. The front and back ends are isolated from each other in separate domains, and communicate solely by mechanisms provided by Xen. The most common of these is the I/O ring, built on top of the shared memory mechanism provided by Xen.

Shared memory rings alone would require a lot of polling, which is not always particularly efficient, although it can be fast where there is pending data in a large percentage of the polled cases. This need is eliminated by the Xen event mechanism, which allows asynchronous notifications. This is used to tell the back end that a request is waiting to be processed, or to tell a front end that there is a response waiting. Handling and delivering events is discussed in the next chapter.

The final part of the jigsaw puzzle is the XenStore. This is a simple hierarchical

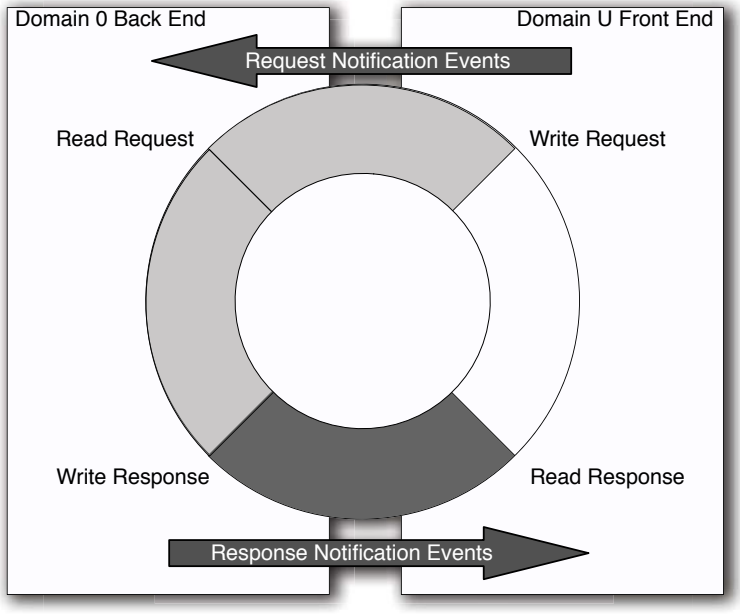

Figure 6.1: The composition of a split device driver

structure that is shared between domains. Unlike the grant tables, the interface is fairly high-level. One of the main uses for it is device discovery. In this rôle, it is analogous to the device tree provided by OpenFirmware, although it has additional uses. The guest in Domain 0 exports a tree containing the devices available to each unprivileged domain. This is used for the initial device discovery phase. The tree is traversed by the guest that wants to run front-end drivers, and any interesting devices are configured. The one exception to this is the console driver. It is anticipated that the console device is needed (or, at the very least, wanted) early on during the boot process, so it is advertised via the start info page.

The XenStore itself is implemented as a split device. The location of the page used to communicate is given as a machine frame number in the start info page. This is slightly different to other devices, in that the page is made available to the guest before the system starts, rather than being exported via the grant table mechanism and advertised in the XenStore.

6.2 Moving Drivers out of Domain 0

Xen provides a mechanism for delegating access to physical devices to domains other than Domain 0, known as *driver domains*. On platforms that don't contain an IOMMU or similar hardware that implements the protection elements of an IOMMU, it is not possible to do this securely.

The hypervisor can use the MMU to isolate the memory regions used by memory mapped I/O and grant pages within these regions to a driver domain. It can also prevent ring 1 processes from using I/O port instructions, and require them to be trapped and emulated by the hypervisor, adding significant overhead. It cannot, however, prevent the driver domain from issuing unsafe DMA requests without additional hardware support. This makes driver domains only semi-safe on most legacy hardware.

The use of driver domains provides two key advantages. The first is that it gives some extra isolation for components of the system. In the standard configuration, Domain 0 has two distinct responsibilities:

- Supporting hardware and running back-end devices
- Providing the administrative interface to Xen

One of the key features of Xen is that only a small amount of code runs in ring 0, which helps provide some security and stability. Code running in ring 1 in Domain 0, however, can still perform a lot of operations via the hypervisor that would usually be restricted to ring 0. By reducing the amount of code running in Domain 0, the security of a system is improved. Device drivers tend to make up the majority of the codebase of a modern operating system and, unfortunately, tend to be the most buggy parts of the kernel. This is understandable, because they are the least tested as not everyone is using the same drivers. Drivers also have to deal with (potentially undocumented) flaws in the physical hardware, as well as the wide range of potential interactions the device can have when functioning correctly. A device domain on a system with an IOMMU can only damage itself with erroneous DMA requests, because the IOMMU prevents accesses to other domains' address spaces.

The other advantage is that it allows support for more devices. If you choose to run Solaris in Domain 0, for example, you are not limited to the devices supported by Solaris. You could run Linux or NetBSD in a driver domain and gain access to other pieces of hardware. This could even be extended by hardware manufacturers to provide a "Xen native" device driver—a minimal kernel with the device driver embedded in it that could be run in a driver domain and reduce the need to provide OS-specific drivers.

6.3 Understanding Shared Memory Ring Buffers

The ring buffer is a fairly standard lockless data structure for producer-consumer communications. The variant used by Xen is somewhat unusual in that it uses free-running counters. A typical ring buffer has a producer and a consumer pointer. Each of these is tested by one and incremented by the other. When one pointer goes past the end of the buffer, it must be wrapped. This is relatively expensive, and because it involves programming for a number of corner cases, it is relatively easy to get wrong.

The Xen version avoids this by ensuring that the buffer size is a power of two. This means that the lowest n bits of the counter can be used to give an index within the buffer, and these bits can be obtained with a simple mask. Because the counters can run to more than the buffer size, you do not need to manually account for overflow; subtracting one from the other always gives you the amount of data in the buffer. The one special case comes from the fact that the counters themselves can overflow. Let's take a simple example—an 8-bit counter on a 32-element buffer—and see what happens when it overflows. The producer value is incremented to 258, which causes an overflow, wrapping the value around to two. The consumer value has not yet overflowed, so it is now bigger than the producer. What happens when we try a subtraction?

	0	0	0	0	0	0	1	0	(Producer value of 2)
−	1	1	1	1	1	1	0	0	(Consumer value of 252)
1	0	0	0	0	0	1	1	0	(Subtraction gives −250)
	0	0	0	0	0	1	1	0	(Truncation gives 6)

The leading 1 in the subtraction result comes from the use of 1's complement arithmetic. If the result of a calculation is negative, the leading bit will be 1, and the remaining bits will be the inverse of their positive values. This representation is used in all modern CPUs, because it simplifies a number of cases (adding signed values can be implemented using the same logic, irrespective of their signs). One nice side effect of this is that subtraction still works if one of the values has overflowed. Note that the leading bit is truncated, because it doesn't fit in the 8-bit value. Most CPUs will store this in a condition register somewhere; however, it can simply be discarded for our purposes.

One thing to note is that this value will be wrong if the producer value overflows twice before the consumer value overflows. Fortunately, this can never happen, because that would require the buffer to be larger than the counter, meaning that no mask exists that would convert the counter value to a buffer index.

The other interesting thing about the Xen ring mechanism is that each ring contains two kinds of data, a request and a response, updated by the two halves of the driver. In principle, this could cause some problems. If the responses are

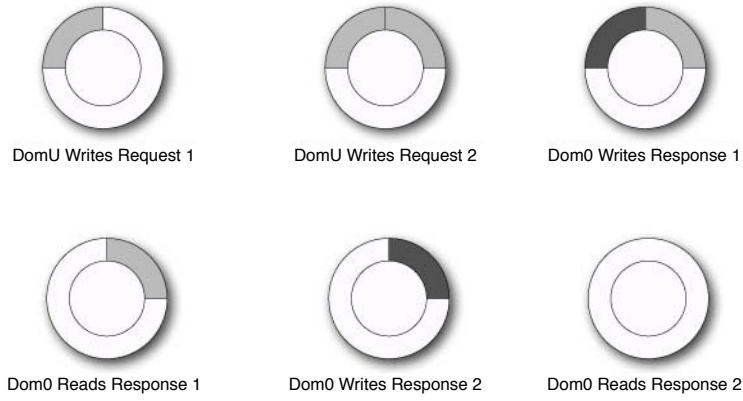

Figure 6.2: A sequence of actions on a ring buffer

larger than the requests, the response pointer catches up with the request pointer, preventing the back end from writing any more responses.

Xen solves this by only permitting responses to be written in a way that overwrites requests. Figure 6.2 shows a typical sequence of actions in an I/O ring. In this sequence, the front half of the driver issues two requests, and the back half fills them in.

The first two operations in this sequence show the front half of the driver writing a pair of requests into the ring. After writing each request, it increments a counter indicating the used part of the ring. The back half then writes a response over the first request. After doing this, it increments the response counter, indicating that the response has been filled in. The front half can then read this response and increment a counter indicating where the back end is. This is then repeated for the second response.

There are three counters of relevance here. The first indicates the start of the requests. This is only ever incremented by the front end, and never decremented. If the back end reads this value, it can guarantee that the value will never be lower than this value.[1] It can then use any space from the back of the request segment to the front in order to store responses.

After storing a response, the back end increments a counter indicating that the response has been stored. The front end can read this counter, and know that anything between it and the back of the tail counter contains responses. It can then read them, and update the tail counter to any value that doesn't cause it to

[1]The only requirement for this to work is that memory writes are atomic. This is true on most modern CPUs. Note that a memory barrier is not required; the old value for the counter is always safe to use, because it is monotonic.

pass the response counter. The tail counter is then used again by the front end, to ensure that it does not overwrite existing requests with new ones.

The ring can only run out of space for responses if there are no outstanding requests, but if this is the case, it will not need to generate any. If it runs out of space for requests, this implies one of two things. Either the ring is full of requests, in which case the front end should back off for a bit and allow the back end to process them, or it contains some responses, in which case, it should consider processing some of them before adding more requests.

6.3.1 Examining the Xen Implementation

Most of the definitions relating to Xen's I/O rings can be found in `xen/interface/public/io/ring.h`. This file contains a number of macros for creation, initialization, and use of I/O rings. This generalized interface is not used by all drivers. Some use a simpler mechanism where there are two unidirectional rings. Others use more complex interfaces. The macros described here are used by both the network and block interfaces, and are suited to any interface that has a one-to-one mapping between requests and responses. This is obviously not the case for the console, where reading from the keyboard and writing to the screen are unrelated operations.

The first macro is used for creating ring structures, in the following way:

```
DEFINE_RING_TYPES(name, request_t, response_t);
```

This creates the structures for a ring where requests were specified by a `request_t` and responses by a `response_t`. This defines three structures representing the ring—one is the shared data structure, which goes in a shared memory page. The other two contain private variables, one for the front end and one for the back. These used by the other macros. Also defined is a union of the request and response types. This is used to divide the rings into segments that can store either a request or a response.

Before a ring can be used, it must be initialized. This sets the producer and consumer indexes to their correct initial values. The macro for initializing the shared ring depends on the existence of `memset`, so ensure that you have a working implementation of this in your kernel, even if it's not properly optimized.

To give some idea of how these are used, we will look briefly at how the virtual block device uses them. The interface to this device is defined in the `io/blkif.h` public header file.

The block interface defines the `blkif_request` and `blkif_response` structures for requests and responses, respectively. It then defines the shared memory ring structures in the following way:

```
DEFINE_RING_TYPES(blkif, struct blkif_request, struct
    blkif_response);
```

Grant Table Use

To use the ring mechanism, both sides of the driver must be able to access it. This means that the underlying memory must be shared, and the standard mechanism for doing this is via the grant table.

The convention for the Xen split driver model is that the front-end driver should offer the grant reference, while the back end maps it. This means that the front end does not need to issue any hypercalls to set up the driver rings. Aside from consistency, this has the advantage that HVM guests can implement paravirtualized device drivers without the need to use any grant table hypercalls. It also aids migration. The shared rings belong to the front end, and are shared with the back end, so when the front end is moved to a different machine they stay in the same place (in the pseudo-physical address space) and can be remapped easily.

This convention applies to all Xen drivers that use the split model, not only those that use the generic ring macros. This means that the only hypercalls that a front-end driver needs to use are the ones that relate to the event channel.

This defines the blkif_sring_t type, representing the shared ring and two other structures representing the private variables used by the front and back halves of the ring. The front end must allocate the space for the shared structure, and then initialize the ring. The Linux implementation of this driver then initializes the ring as follows:

```
SHARED_RING_INIT(sring);
FRONT_RING_INIT(&info->ring, sring, PAGE_SIZE);
```

Here, the sring variable is a pointer to the shared ring structure. The info variable is a pointer to a structure that contains some kernel-specific bookkeeping information about the virtual block device. The ring field is a blkif_front_ring_t, which is a structure defined by the ring macros to store the private variables relating to the front half of the ring. These are required due to the way in which data is put on the ring.

The first step is to write the request to the ring. Then, the (shared) ring's request producer count is incremented. For efficiency, it is fairly common to write multiple requests into the ring, and then perform the update. For this reason, it is necessary to keep a private copy of the request producer counter, which is incremented to let the front end know where it should put new requests. This is then pushed to the shared ring later. The next available request in the ring is requested by the front end like this:

```
ring_req = RING_GET_REQUEST(&info->ring, info->ring.
    req_prod_pvt);
```

The request is then filled in with the correct information, and is ready to be pushed to the front end. This is done using the following macro:

```
RING_PUSH_REQUESTS_AND_CHECK_NOTIFY(&info->ring, notify);
```

Here, notify is a return parameter, which is used to indicate whether the front end needs to deliver an event to the back end. This macro sets the shared ring's request producer counter to be equal to the private copy. After doing this, it tests the request consumer counter. If this is still lower than the request producer counter value from before the update, it means that the back end is still processing requests that were enqueued before this update was pushed. If this is the case, there is no need to send an event to the back end, and so notify is set to false.

After a request has been sent, the back end processes it and inserts a response into the ring. The next response from the ring is fetched like this:

```
blkif_response_t *bret = RING_GET_RESPONSE(&info->ring, i);
```

This sets bret to the response at index i in the ring. The value of i needs to be somewhere between the shared ring's rsp_prod and the front half's rsp_cons values. These two fields represent the response producer and consumer indexes. After processing response i, the front half's rsp_cons value should be updated to reflect this. This value is used when inserting new requests into the ring, because it indicates where the back of the used segment of the ring is. No new requests may be inserted past this value. After fetching all waiting requests, the block driver performs the following check:

```
RING_FINAL_CHECK_FOR_RESPONSES(&info->ring, more_to_do);
```

The more_to_do value is set to true if the ring still has some requests waiting to process. After checking this, the event channel should be unmasked, the ring checked one final time, and then control returned.

The back end has a similar interface, but uses the other private structure, and uses the queues the other way around.

6.3.2 Ordering Operations with Memory Barriers

Some operations on rings require *memory barriers* on some systems. On a purely in-order architecture, memory operations are guaranteed to be completed in the order in which they are issued. This means that, as long as the data is written into the ring before the counters are incremented, things will work correctly.

x86 began life as an in-order architecture. As it has grown, it has had to maintain compatibility with legacy code. For this reason, modern x86 CPUs are still *strongly ordered*. Memory operations on a strongly ordered CPU are

guaranteed to complete in the order in which they are issued. On x86, this is only true for stores; loads may be reordered.

This is not true of all architectures supported by Xen. IA64, for example, is a *weakly ordered* architecture, and so no such guarantees exist. This is somewhat complicated by the fact that early versions of the Itanium were strongly ordered. This means that code written and tested on an early Itanium might suddenly fail on a newer version.

To prevent re-ordering, most architectures provide memory barrier instructions. These force all loads, stores, or loads and stores to be completed before continuing. The Xen code uses macros to implement these, and expands them based on a per-processor definition. When developing on x86, it is tempting to leave some of these out, because the write barrier expands to a no-op on x86. If you do this, be aware that your code will break on other platforms.

The Xen ring macros include the correct barriers, and work on all supported architectures. For other devices that don't use these macros, it is important to remember to add the barriers yourself. The most commonly used barrier macros are wmb() and mb(), which provide a write memory barrier and a full (read and write) barrier, respectively. Here is an example of a barrier being used in one of the standard ring macros:

```
#define RING_PUSH_REQUESTS(_r) do {                                    \
    wmb(); /* back sees requests /before/ updated producer
        index */      \
    (_r)->sring->req_prod = (_r)->req_prod_pvt;                        \
} while (0)
```

The wmb() macro is used here to ensure that the requests that have been written to the ring are actually in memory before the private copy of the request producer counter is copied into the shared ring. A full memory barrier is used on the macros that check whether they need to notify the other end of new data, to ensure that any reads of the data in the ring by the remote end have completed before checking the consumer counter.

Note that x86 has no explicit barrier instructions. Write barriers are not needed, but read barriers are on some occasions. Instructions with the **LOCK** prefix are implicit read barriers, and so an atomic add instruction is used in place of an explicit barrier. This means that barriers can be omitted on x86 when they immediately follow an atomic operation.

6.4 Connecting Devices with XenBus

The XenBus, in the context of device drivers, is an informal protocol built on top of the XenStore, which provides a way of enumerating the (virtual) devices available to a given domain, and connecting to them. Implementing the XenBus interface is not required when porting a kernel to Xen. It is predominantly used in Linux to isolate the Xen-specific code behind a relatively abstract interface.

The XenBus interface is intended to roughly mirror that of a device bus such as PCI. It is defined in `linux-2.6-xen-sparse/include/xen/xenbus.h`.[2] Each virtual device has three major components:

- A shared memory page containing the ring buffers
- An event channel signaling activity in the ring
- A XenStore entry containing configuration information

These components are tied together in the bus interface by the structure shown in Listing 6.1. This device structure is passed as an argument to a number of other functions, which (between them) implement the driver for the device.

Listing 6.1: The structure defining a XenBus device [from: linux-2.6-xen-sparse/include/xen/xenbus.h]

```
71 struct xenbus_device {
72     const char *devicetype;
73     const char *nodename;
74     const char *otherend;
75     int otherend_id;
76     struct xenbus_watch otherend_watch;
77     struct device dev;
78     enum xenbus_state state;
79     struct completion down;
80 };
```

The exact definition of this structure is tied quite closely to Linux; the device struct, for example, represents a Linux device. It can, however, be used as a good starting point for building a similar abstraction layer for other systems.

The core component of the XenBus interface, indeed the only part that needs to be implemented by all systems wanting to use the paravirtualized devices available to Xen guests, is the xenbus_state enumerated type. Each device has such a type associated with it.

[2] Although the header is part of the sparse Linux tree, it is available under a more permissive license when not distributed as part of Linux.

The XenBus state, unlike the rest of the XenBus interface, is defined by Xen, in the `io/xenbus.h` public header. This is used while negotiating a connection between the two halves of the device driver. There are seven states defined. In normal operation, the state should be gradually incremented as the device is initialized, connected, and then disconnected. The possible states are:

- XenbusStateUnknown represents the initial state of the device on the bus, before either end has been connected.
- XenbusStateInitialising is the state while the back end is in process of initializing itself.
- XenbusStateInitWait should be entered by the back end while it is waiting for information before completing initialization. The source of the information can be hot-plug notifications within the Domain 0 kernel, or further information from the connecting guest. The meaning of this state is that the driver itself is initialized, but needs more information before it can be connected to.
- XenbusStateInitialised should be set to indicate that the back end is now ready for connection. After the bus is in this state, the front end may proceed to connect.
- XenbusStateConnected is the normal state of the bus. For most of the duration of a guest's run, the bus will be in this state indicating that the front and back ends are communicating normally.
- XenbusStateClosing is set to indicate that the device has become unavailable. The front and back halves of the driver are still connected at this point, but the back end is no longer doing anything sensible with the commands received from the front. When this state is entered, the front end should begin a graceful shutdown.
- XenbusStateClosed is the final state, once the two halves of the driver have disconnected from each other.

Not all drivers make use of the XenBus mechanism. The two notable exceptions are the console and XenStore. Both of these are mapped directly from the start info page, and have no information in the XenStore. In the case of the console, this is so that a guest kernel can start outputting debugging information to the console as soon as possible. In the case of the XenStore, it is obvious that the device can't use XenBus, because XenBus is built on top of the XenStore, and the XenStore cannot be used to get information required to map itself.

6.5 Handling Notifications from Events

Real hardware uses interrupt channels to notify the CPU of asynchronous events. These then cause the CPU to enter privileged mode and jump to a handler that has been configured for the event.

Xen provides an analog of this in the form of event channels. These are delivered asynchronously to guests and, unlike interrupts, are enqueued when the guest is not running. Interrupts, conventionally, are not aware of the existence of virtual machines and so are delivered immediately.[3]

When a driver needs to notify a device of some waiting data, it typically writes to a control register. Within Xen, the event mechanism also replaces this for notifying the back end of a split device driver, because both directions are an example of interdomain communication.

Events have a couple of major differences from interrupts. They are bidirectional and connection-oriented. An IRQ is delivered to a specified handler, but the concept of a connection does not arise. What can raise an interrupt is defined at the hardware level; the interrupt descriptor table indicates how privileged a program needs to be to trigger a given interrupt in software, and the hardware defines which devices can trigger them externally.

An event needs much better access control. A malicious guest could cause some serious problems by triggering large numbers of spurious events. For this reason, events may only be delivered on a given channel by one of the two domains to which it is connected. When an event channel is allocated by one domain, it explicitly states the number of the domain that is allowed to bind to the other end. The other domain must then explicitly request binding to that event channel. It is not until this point that either end may trigger the event delivery.

The event channel number for the device must be passed to the front end somehow, before it can connect.

In addition to providing asynchronous notifications sent from one domain to another, events can be used by a guest to receive real IRQs. These cannot be delivered using the normal mechanism because, as mentioned earlier, they might well end up being delivered to the wrong VM. The hypervisor must catch interrupts raised by devices and enqueue them so that the domain hosting the back end of the driver can receive them irrespective of which domain was running when the interrupt was generated.

Event channels are also used to deliver notifications from a (small) number of devices that run inside the hypervisor. The most common of these is the domain virtual time clock device. This is typically used for scheduling; it provides a notification when a certain amount of virtual time has elapsed, that is, when the domain has received $n$ms of CPU time.

[3] Modern, virtualization-aware hardware provides a mechanism for enqueuing interrupts.

6.6 Configuring via the XenStore

Among other things, the XenStore is the Xen equivalent of an OpenFirmware device tree, or the results of querying an external bus. It provides a central location for retrieving information about devices that are available to the domain. The XenStore is, itself, a device, and so must be bootstrapped using information from the start info page. Chapter 8 will discuss this process in more detail, and give a more comprehensive overview of the XenStore.

Each virtual machine has an entry in the XenStore in which all information about that VM is stored. This is true for both the front and back ends of the device. As mentioned earlier, the back ends are not always in Domain 0, so the front end needs to be able to know three things when connecting to a typical Xen device:

- The domain hosting the back end
- The grant reference of the shared memory page
- The event channel used for notifications

It may also need to know some other device-specific information. This could be passed in the shared memory page, but passing it in the store allows it to be inspected before the device is initialized, and allows extra information to be added without breaking the device's ABI. It also allows tools to access the information about the device, for various reasons.

The XenStore provides an abstract way of discovering information about devices, and about other aspects of the system. It is one of the first devices that must be supported, because it allows the information required for other devices to be found.

The XenStore is a simple hierarchical namespace containing strings. This eliminates a number of potential problems. When running x86 and x86-64 guests on the same machine, word size becomes an issue for binary interfaces. An even worse situation can occur on some architectures, such as PowerPC, ARM, or SPARC, which are bi-endian. Big-endian and little-endian guests could be running on the same system; passing anything other than a string of characters (bytes) would require careful use of things like the `htonl` macros to ensure consistent storage. Another advantage of the text-based nature of the XenStore is that it makes it much easier for tools written in scripting languages to parse the data.

6.7 Exercise: The Console Device

The console device is simpler than many of the others. Instead of using the generic ring mechanism, where a single ring contains requests and responses, it provides

two rings containing input and output characters, respectively. This is because console interaction on the dumb terminal model is intrinsically composed of two unidirectional systems. The keyboard writes data into the system in response to the user, whereas the screen displays text without providing any information.

The console interface itself is very simple. Listing 6.2 describes the structure found on the shared memory page used by the console. The machine address of this page is passed to the guest in the start info structure.

Listing 6.2: Xen Console interface structure [from: xen/include/public/io/console.h]

```
34 struct xencons_interface {
35     char in[1024];
36     char out[2048];
37     XENCONS_RING_IDX in_cons, in_prod;
38     XENCONS_RING_IDX out_cons, out_prod;
39 };
```

Before the console can be used, the guest needs to map it into its address space. This is done as per the example in Chapter 5. After this has been accomplished, the event channel for console events must be bound. This will be covered in more detail in the next chapter; for now, we will treat the console as a write-only device, and try to display a boot message.

Listing 6.3 shows how to map the console. We will leave some space here for setting up the event handler, and come back to that in the next chapter, after detailed discussion of the events system. We will keep a record of the event channel that is being used for the console in the console_evtchn variable.

Listing 6.3: Mapping the console [from: examples/chapter6/console.c]

```
10 /* Initialise the console */
11 int console_init(start_info_t * start)
12 {
13     console = (struct xencons_interface*)
14         ((machine_to_phys_mapping[start->console.domU.mfn] <<
              12)
15          +
16         ((unsigned long)&_text));
17     console_evt = start->console.domU.evtchn;
18     /* TODO: Set up the event channel */
19     return 0;
20 }
```

When we want to write something to the screen, we have to copy it into the buffer, assuming that there is space. If there is insufficient space, we have to wait until there is. Typically, copying a load of data would be done using memcpy. Because we are not linking against the C standard library, however,

this is not an option for us. Instead, we have to implement the copy operation ourselves. The version shown here is not at all optimized; it is possible to make it significantly faster. This is not particularly important for a console driver, because the console is typically a fairly low data rate device; however, for other drivers, it is probably worth copying a well-optimized memcpy implementation, assuming you don't already have one in your kernel.

Listing 6.4 contains a simple function for writing a string to the console. The function loops for each character on an input string until it encounters a NULL, copying the character from the input string to the buffer.

Listing 6.4: Writing data to the console [from: examples/chapter6/console.c]

```
22 /* Write a NULL-terminated string */
23 int console_write(char * message)
24 {
25     struct evtchn_send event;
26     event.port = console_evt;
27     int length = 0;
28     while(*message != '\0')
29     {
30         /* Wait for the back end to clear enough space in the
               buffer */
31         XENCONS_RING_IDX data;
32         do
33         {
34             data = console->out_prod - console->out_cons;
35             HYPERVISOR_event_channel_op(EVTCHNOP_send, &event);
36             mb();
37         } while (data >= sizeof(console->out));
38         /* Copy the byte */
39         int ring_index = MASK_XENCONS_IDX(console->out_prod,
               console->out);
40         console->out[ring_index] = *message;
41         /* Ensure that the data really is in the ring before
               continuing */
42         wmb();
43         /* Increment input and output pointers */
44         console->out_prod++;
45         length++;
46         message++;
47     }
48     HYPERVISOR_event_channel_op(EVTCHNOP_send, &event);
49     return length;
50 }
```

Note the use of the MASK_XENCNOS_IDX macro. This is used because the console rings, like other I/O rings in Xen, use free-running counters. The least-significant n bits of these are used to indicate the position within the ring. The advantage of this approach is that there is no need to test whether the counters have overflowed the buffer. In addition, comparisons of the form *producer* − *consumer* will always give the correct result, as long as the size of the maximum value of the index variable is greater than twice the size of the buffer, and the *producer* counter doesn't overflow twice before the *consumer* counter overflows once.

This means that we only need to test for one condition before adding something to a ring—that $producer - consumer < ring\ size$. Because $producer - consumer$ always gives you the amount of data in the ring, this does not allow you to proceed if the ring is full. In a full implementation of the console, we would wait until an event is received before proceeding here.

After putting data in the buffer, we need to signal the back end to remove it and display it. This is done via the event channel mechanism. Events will be discussed in detail in the next chapter, including how to handle incoming ones. For now, we will just signal the event channel, and look at what is actually happening in the next chapter.

Signaling the event channel is fairly simple, and can be thought of in the same way as sending a UNIX signal. The console_evtchn variable holds the number of the event channel being used for the console. This is similar to the signal number in UNIX, but is decided at runtime, rather than compile time. Note that sending an event after each character has been placed into the ring is highly inefficient. It is more efficient to only send an event at the end of sending, or when the buffer is full. This is left as an exercise for the reader.

To issue the signal, we use the HYPERVISOR_event_channel_op hypercall. The command we give to this tells it to send the event, and the control structure takes a single argument, indicating the event channel to be signaled.

We will add one more function for our simple half-console driver. This flushs the output buffer by blocking until the buffer is empty (that is, the consumer counter has caught up with the producer in the output ring). Because the function is just spinning while waiting for the back end to catch up, we issue a hypercall to notify the hypervisor that it should schedule other virtual machines while we are waiting. The memory barrier here is almost certainly not needed, because a hypercall (and the resulting ring transition) is a memory barrier, but is left in for clarity. Listing 6.5 shows the flush function.

Now that we have something that is hopefully a working implementation of a write-only console driver (we are not reading text input by the user yet), we should try using it. Let's create a console.h file that contains prototypes to our two functions, and then try calling them. Listing 6.6 gives the body of a kernel that writes "Hello world" to the console. "Hello world" is the obligatory first

Listing 6.5: Flushing the console output buffer [from: examples/chapter6/console.c]

```
52 /* Block while data is in the out buffer */
53 void console_flush(void)
54 {
55     /* While there is data in the out channel */
56     while(console->out_cons < console->out_prod)
57     {
58         /* Let other processes run */
59         HYPERVISOR_sched_op(SCHEDOP_yield, 0);
60         mb();
61     }
62 }
```

output message from any system, but it's a bit boring. Let's print the Xen magic string, containing the running Xen version, as well.

Note that we call the console_flush() function before exiting. This is because the console ring buffer ceases to exist when the domain is destroyed, and if we are not very lucky, this will happen before the back end has read the contents of the buffer.

All that remains now is to add `console.o` to the `Makefile`, build, and test our kernel. When we launched our last simple kernel, we used `xm create`. This creates the new domain in the background. When we start this one, we want to see the output from the console. We do this by adding the `-c` flag, which tells `xm` to automatically attach to the console:

```
# xm create -c domain_config
Using config file "./domain_config".
Started domain Simplest_Kernel
Hello world!
Xen magic string: xen-3.0-x86_32p
#
```

The new domain is then destroyed, because we told the kernel to exit after writing the message. If this happens, everything is working as expected. We can now use the console for output during the rest of our boot procedure. When we have mapped the event channel, we can use it for input as well.

Currently, the console output is a little bit raw. It would be nice to add something like the C standard printf() function. This is left as an exercise to the reader. A good approach is to take a look at the vfprintf() function in an existing C library (the OpenBSD implementation is a good bet here) and replace the function or macro used for actually outputting characters with a call to our console_write() function.

Listing 6.6: The body of the "hello world" kernel [from: examples/chapter6/kernel.c]

```
12 /* Main kernel entry point, called by trampoline */
13 void start_kernel(start_info_t * start_info)
14 {
15     /* Map the shared info page */
16     HYPERVISOR_update_va_mapping((unsigned long) &shared_info,
17             __pte(start_info->shared_info | 7),
18             UVMF_INVLPG);
19     /* Set the pointer used in the bootstrap for reenabling
20      * event delivery after an upcall */
21     HYPERVISOR_shared_info = &shared_info;
22     /* Set up and unmask events */
23     init_events();
24     /* Initialise the console */
25     console_init(start_info);
26     /* Write a message to check that it worked */
27     console_write("Hello world!\r\n");
28     /* Loop, handling events */
29     while(1)
30     {
31         HYPERVISOR_sched_op(SCHEDOP_block,0);
32     }
33 }
```

This page intentionally left blank

Chapter 7

Using Event Channels

Event channels are the primitive mechanism for asynchronous notifications within Xen. They are used in conjunction with ring buffers in shared memory pages to provide an efficient message passing mechanism for communicating between the front and back ends of a split device driver.

This chapter will explore the similarities and differences between Xen events and software and hardware interrupts. We will look at how event channels are bound and how the end points are used, learning both how to signal an event channel, and handle the event that is raised.

7.1 Events and Interrupts

Events are the standard mechanism for delivering notifications from the hypervisor to guests, or between guests. Conceptually, they are similar to (traditional) UNIX signals. Each event delivers one bit of information: that the event has occurred.

The standard way in which a signal is delivered is via an upcall from the hypervisor. As with signals, an event can be delivered while another one is being handled. It is therefore common to disable event delivery during a handler.

Unlike UNIX signals, events that occur while delivery is disabled are not lost. They are not delivered after the handler is reenabled; however, it is possible to check for existing events by polling the relevant *Virtual CPU* (*VCPU*) structure in the shared info page.

In many ways, Xen events replace hardware interrupts. An interrupt is an asynchronously delivered trigger that something has happened related to the machine hardware. An event is an asynchronously delivered trigger that something has happened related to the virtual machine. Although a real network card might

send a signal every time a packet is received, a virtual interface might send an event.

7.2 Handling Traps

In addition to events, Xen provides a lower-level form of asynchronous notification in the form of traps. Unlike events, which can be dynamically created and bound, traps have static meanings, corresponding directly to hardware interrupts.

The IA32 specification breaks interrupts down into three broad categories: used, unused, and reserved. The first 20 are all used except for 1, 9, and 15, which are reserved.[1] The next 11 are reserved, and the remainder are all unused. Of the unused ones, 80h is typically used for system calls, and 82h was used for hypercalls in earlier versions of Xen.

The code path for delivering a trap is significantly simpler than that for events. When the guest is run on a particular (physical) CPU, the hypervisor installs an *Interrupt Descriptor Table* (*IDT*) on behalf of the guest domain. This means that the interrupt handling path does not involve the hypervisor at all, for all interrupts are handled by the guest.

Because Xen traps correspond directly to hardware interrupts, the same code can typically be reused to handle them. For example, a floating point error should either cause a signal to be raised in the running process, the process to be terminated, or the signal simply to be ignored, depending on the operating system or user's standard behavior. This is no different under Xen. The one change is the way of accessing certain control registers. While an interrupt handler is running, with interrupts disabled, it is completely guaranteed that nothing else will touch the CPU state. This is not true on Xen, where a virtual machine might be preempted to allow another to run.[2] Because certain registers on x86 can't be written to directly, they will not have their contents refilled when the guest next runs. To prevent information being lost, the hypervisor stores the values of fragile control registers in the virtual CPU structure in the shared info page. On x86, the **CR2** register, containing the address of the last page fault, is handled in this way.

The most obvious reason a guest might want to do this is to handle interrupt 80h directly. A Xen guest can't use **SYSCALL** or **SYSENTER** for (fast) system calls, because these jump directly to ring 0, where the hypervisor is resident, and must be reflected back to the guest kernel. Instead, the more traditional interrupt

[1] More accurately, 9 is no longer used. It is a historical artifact and is no longer generated by modern x86 chips.

[2] In theory, it is possible to prevent this by disabling preemption in the hypervisor while an interrupt handler was running. In practice, however, this would leave other guests open to a denial of service attack by a malicious guest that ran the entire kernel from an interrupt handler.

80h method is commonly used. By installing a trap vector for interrupt 80h, the guest can handle system calls without the hypervisor having any involvement.

Fast System Calls and HVM

Unlike paravirtualized guests, those running in a hardware virtual machine domain do not experience ring compression. The kernel of an HVM guest runs in ring 0, with the hypervisor in a new mode, hidden from ring 0. This means that **SYSENTER** and **SYSCALL**, which provide a fast transition to ring 0, can be used for system calls.

At the time of writing, paravirtualized guests do not run in HVM mode, and so are unable to take advantage of this. In future releases, it is likely that more HVM features will be added to PV domains, making this possible for both kinds of guests when the underlying hardware supports HVM.

The IDT installed by this hypercall is removed when the guest is not running. For this reason, it should only be used to receive interrupts that will be caused by things running in the domain. These include things like processor exceptions, where the kernel only cares if they are caused by processes for which it is responsible. They should not be used for things like receiving interrupts from devices, because this will cause interrupts to be lost when the VM is not scheduled.

The structure of the trap table is not the same as the IDT. Early versions of Xen did not attempt to mimic the structure of the host platform; they aimed to provide a fairly generic abstraction layer. For this reason, the trap table is a simple structure that can be easily manipulated by the guest, as shown in Listing 7.1.

Listing 7.1: Trap table entry for x86 [from: xen/include/public/arch-x86/xen.h]

```
101 struct trap_info {
102     uint8_t       vector;  /* exception vector
                                  */
103     uint8_t       flags;   /* 0-3: privilege level; 4: clear
            event enable? */
104     uint16_t      cs;      /* code selector
                                    */
105     unsigned long address; /* code offset
                                      */
106 };
```

Each entry contains the number of the trap, the highest number privilege ring that can raise the interrupt in software, and the address of the trap handler. The trap number is the same as the interrupt number on the host platform. The privilege level defines a ring number. This has no real meaning on x86-64,

because both the kernel and userspace applications live in ring 3. On 32-bit x86, however, rings 1-3 are available for the guest domain to use. The kernel typically lives in ring 1, and userspace applications in ring 3. Some kernel designs demote certain privileged components to ring 2. It might be necessary for these to raise interrupts that are handled by the ring 1 kernel component. In this case, you set the second parameter to 2, which allows the privileged components to execute an **INT** instruction and pass control to the ring 1 component, but prevents the same from happening from a userspace application.

Listing 7.2 shows the trap table initialization code from the Mini-OS example distributed with Xen. The last two fields in each entry identify the handler as a segment number and the offset within that segment. The __KERNEL_CS symbol is set to FLAT_KERNEL_CS, which is defined by Xen. This represents the code segment created by Xen mirroring a flat address space, where the entire address space is mapped into a single segment.

Listing 7.2: Trap table initialization code from the Xen Mini-OS

```
202 static trap_info_t trap_table[] = {
203  { 0, 0, __KERNEL_CS, (unsigned long)divide_error        },
204  { 1, 0, __KERNEL_CS, (unsigned long)debug               },
205  { 3, 3, __KERNEL_CS, (unsigned long)int3                },
206  { 4, 3, __KERNEL_CS, (unsigned long)overflow            },
207  { 5, 3, __KERNEL_CS, (unsigned long)bounds              },
208  { 6, 0, __KERNEL_CS, (unsigned long)invalid_op          },
209  { 7, 0, __KERNEL_CS, (unsigned long)device_not_available},
210  { 9, 0, __KERNEL_CS, (unsigned long)
         coprocessor_segment_overrun},
211  {10, 0, __KERNEL_CS, (unsigned long)invalid_TSS         },
212  {11, 0, __KERNEL_CS, (unsigned long)segment_not_present },
213  {12, 0, __KERNEL_CS, (unsigned long)stack_segment       },
214  {13, 0, __KERNEL_CS, (unsigned long)general_protection  },
215  {14, 0, __KERNEL_CS, (unsigned long)page_fault          },
216  {15, 0, __KERNEL_CS, (unsigned long)spurious_interrupt_bug},
217  {16, 0, __KERNEL_CS, (unsigned long)coprocessor_error   },
218  {17, 0, __KERNEL_CS, (unsigned long)alignment_check     },
219  {19, 0, __KERNEL_CS, (unsigned long)simd_coprocessor_error},
220  { 0, 0,           0, 0                                  }
221 };
222
223 void trap_init(void)
224 {
225     HYPERVISOR_set_trap_table(trap_table);
226 }
```

Each address in this table is an entry point written in assembly that saves the registers in a pt_regs structure on the stack and then calls a C function that

performs the actual handling. This is similar in structure to the event handler entry point, described in detail later in this chapter.

Non-x86 Traps

On most non-x86 platforms, the Xen port uses the native support for virtualization. In these cases, the guest can simply install an IDT as it would normally. This is true on PowerPC and IA64. Other architectures, such as ARM, might retain the trap table mechanism, but provide a different trap structure. At the time of writing, the only platform for which the trap mechanism has any meaning is x86. Because the trap_info_t structure is defined on a per-platform basis, it is likely that future ports wanting to use it would use the structure of the underlying platform's IDT, rather than create a new structure.

7.3 Event Types

Events fall into three broad categories; *interdomain events*, *physical IRQ*, and *virtual IRQs* (*VIRQ*). Physical IRQs are, perhaps, the easiest to understand. They are mappings of real IRQs to event channels. Unlike traps, events will be enqueued even when the domain is not scheduled, and then delivered when it is. For this reason, they should be used for communicating with hardware devices.

A guest in Domain 0, or in a driver domain, will want to set up physical IRQ to event channel mappings for the various devices under its control. Before doing this, of course, it will want to discover which devices are already bound to which IRQs. Typically, this is done via BIOS or APIC calls. This is not permitted in Xen, however, so they are forced to use the HYPERVISOR_physdev_op hypercall.

The second kind of events are virtual IRQs. These are similar to physical IRQs, but related to virtual devices. The most primitive example of this is the timer. The timer virtual device is similar to its physical counterpart, but exists in domain virtual time. The guest can request a timer event at a specified virtual time, at which point it receives an event on the channel bound to the VIRQ_TIMER virtual IRQ.

There are some other virtual IRQs associated with hypervisor-provided devices, but most of these are only usable from within Domain 0. Mostly, these relate to debugging guest domains. One example of these is VIRQ_CONSOLE, which notifies the Domain 0 guest that a guest has written some data to the debug console.[3]

[3]The debug console is only available to other domains if the hypervisor is compiled with debugging support.

The final category, interdomain events, are more fuzzily defined. These are created as a two-stage process. One domain allocates a new event channel as an unbound channel, and grants permission for the other domain to bind to it. The second domain then allocates a new channel and binds it to the remote domain's port. When the connection is complete, either domain may signal the other by sending an event to the local port. The main use of interdomain events is for what could be thought of as "paravirtual IRQs." These are interrupts caused by paravirtualized devices. Most devices use these to notify the guest domain that there is data waiting. Unlike IRQs, they are bidirectional. The guest domain sends an event to notify the guest that there is data waiting to be transported in the opposite direction.

Although it is unusual, there is a fourth category: *intradomain events*. These are really a special case of interdomain events where both the sending and receiving domain are the same, the equivalent to *interprocessor interrupts* (*IPIs*) in a physical system. It is possible to use event channels to communicate between VCPUs in the same guest. There are often better ways of doing this, but like UNIX domain socket programming in userspace, it has the advantage that it can later be extended to external domains. If you have a feature in your operating system that might be better isolated in another domain in the future for extra security, event channels might be a good choice.

Be aware when communicating between VCPUs that there is no guarantee that they will be running concurrently. One physical CPU might be used to schedule all of your configured VCPUs. If you have two VCPUs in a producer / consumer relationship, make sure you issue a "yield" scheduling operation to the hypervisor when one is waiting to allow the other to run.

7.4 Requesting Events

Requesting events is a fairly simple procedure. All that is required is to bind an event channel to an event source—either a real or virtual IRQ, or a remote domain's event channel. The process can be broken down into two stages:

1. Binding the channel to an event source
2. Configuring a handler for the event

The first step is the process of actually binding the channel to an event source. One of the first event channels that a kernel needs to bind is the timer virtual interrupt. This is identified by VIRQ_TIMER, which provides a periodic timer event that can be used for scheduling.

Listing 7.3 shows how the timer interrupt can be bound. The hypervisor selects an unbound event channel in the caller, and sets the value of op.port to this value. This should then be used in the second step, where the handler is configured.

Channels and Ports

The terms "channel" and "port" are used almost interchangeably. Technically, a channel is the abstract connection between two endpoints, whereas the port is an identifier used to indicate which endpoint the channel is connected to.

When an interdomain event channel is allocated, it is connected to a port in the domain that creates it. When a remote domain binds it, both ends have a port and the channel can be used. From the perspective of either domain, the local port is the channel; it has no other means of identifying it. It is important to be aware of the distinction between channels and ports, although in practice it is safe to ignore it most of the time.

Listing 7.3: Binding the timer virtual interrupt

```
1 evtchn_bind_virq_t op;
2
3 op.virq = VIRQ_TIMER;
4 op.vcpu = 0;
5
6 if ( HYPERVISOR_event_channel_op(EVTCHNOP_bind_virq, &op) != 0
      )
7 {
8     /* Handle the error */
9 }
```

The other kind of event that is commonly requested is the interdomain event. This is used for delivery of events between the two halves of split drivers, and can be used for signalling other asynchronous interdomain communication events. The process for binding these events is similar to that of binding VIRQs. Note that not all events of this nature need to be bound; the event channels used for console and the XenStore are bound by the domain builder. The console event channel is bound so that it can be used early on for outputting kernel debugging information. The XenStore event channel is bound so that the XenStore can be used for bootstrapping other interdomain communication.

Virtual IRQs originate inside the hypervisor, and so the only information required to bind one is the VIRQ number. Interdomain events must be bound to a remote domain and an allocated (but unbound) channel within that domain.

Listing 7.4 shows how a new event channel can be allocated. As before, op.port is set to the allocated port. This operation allocates a channel that remote_domain may bind for interdomain communication. The value of op.port usually is placed in

Listing 7.4: Allocating an unbound event channel

```
1 evtchn_alloc_unbound_t op;
2 op.dom = DOMID_SELF;
3 op.remote_dom = remote_domain;
4 if( HYPERVISOR_event_channel_op(EVTCHNOP_alloc_unbound, &op) !=
      0)
5 {
6     /* Handle the error */
7 }
```

the XenStore, where the remote domain can access it. Only the domain specified when the channel is allocated is allowed to bind to it.

Assuming that the other domain has now managed to retrieve the event channel port number via some means (typically the XenStore), it can now bind it for interdomain access, as shown in Listing 7.5.

Listing 7.5: Binding an event channel for interdomain communication

```
1 evtchn_bind_interdomain_t op;
2 op.remote_dom = remote_domain;
3 op.remote_port = remote_port;
4 if( HYPERVISOR_event_channel_op(EVTCHNOP_bind_interdomain, &op)
      != 0)
5 {
6     /* Handle the error */
7 }
```

There are a few caveats to add to this related to masking events, which are discussed in the next section. It is common at boot time to mask all event channels, and then unmask them as they are bound, to remove the possibility of receiving spurious events. When binding a new event, you should also clear the pending bit relating to that event. Without doing this, it is possible for the handler to be triggered when it shouldn't be. The final thing to note is that event delivery is disabled at boot time, and needs to be enabled by clearing the VCPU's evtchn_upcall_mask flag. After doing this, you must check whether the VCPU's evtchn_upcall_pending flag is set, and handle waiting events if it is. Failing to do so can result in events being lost or delayed.

Intradomain events, the virtualized form of interprocessor interrupts, are a third kind of event channel. These are slightly different from other event channels. Because both endpoints are in the same domain, the port number is the same for both ends. Very little information needs to be passed to create such an event channel. The domain is implicit—it must be the local domain—and the port for both ends is allocated by the hypercall. The virtual CPU at one end is implicitly

set to the caller's VCPU. All that is required is the VCPU at the other end of the channel, as shown in Listing 7.6.

Listing 7.6: Binding an event channel for intradomain communication

```
1 evtchn_bind_ipi op;
2 op.vcpu = other_vcpu;
3 if( HYPERVISOR_event_channel_op(EVTCHNOP_bind_ipi, &op) != 0)
4 {
5     /* Handle the error */
6 }
```

The final kind of event that can be handled is a physical IRQ. As discussed earlier, these cannot be delivered directly to running guests[4], and so must be routed through the hypervisor.

Unlike the other event types, most domains are not permitted to request physical IRQs. Domain 0 is typically allowed to, as is any driver domain that is explicitly configured to have access to a given IRQ.

The second step in the binding process does not involve any hypervisor interaction, and occurs entirely within the guest. All events are delivered to a single event handler in the guest kernel, which is responsible for dispatching them to the correct location. Before an event channel is set up for handling incoming events, it is necessary to set something in this handler so it knows how to handle the new event. One way of doing this will be discussed in more detail in the example at the end of the chapter.

7.5 Binding an Event Channel to a VCPU

For uniprocessor guests, events are passed to the only VCPU. For SMP guests, it is likely that the handler for a given event may want to be bound to a particular VCPU. This is more useful on nonvirtualized guests, because it enables better use of the processor's instruction cache, but it can be useful on virtualized systems to allow cheaper locking and more sensible scheduling decisions.

Only interdomain events can be bound to a VCPU. Virtual IPIs are bound to a pair of virtual CPUs when the channel has been bound, and per-VCPU VIRQs are similarly bound to a given VCPU. Interdomain events are bound to VCPU 0 when they are created, but can later be rebound to another VCPU.

Listing 7.7 shows how the event channel identified by event_channel_number would be assigned to the VCPU identified by vcpu_number. Note the somewhat confusing terminology in the hypercall name. In various places throughout the Xen code and documentation, the verb "to bind" has three distinct meanings when related to event channels:

[4]Except on some platforms with a virtualization-aware interrupt controller.

Listing 7.7: Assigning a bound event channel to a VCPU

```
evtchn_bind_vcpu op;
op.port = event_channel_number;
op.vcpu = vcpu_number;
if( HYPERVISOR_event_channel_op(EVTCHNOP_bind_vcpu, &op) != 0)
{
    /* Handle the error */
}
```

- Connecting the endpoints of a channel to a port
- Assigning a VCPU for receiving events on a given channel
- Setting the handler for a specified event

The third use is not found in the hypervisor itself (because it occurs inside a guest kernel), but is found in the example Mini-OS kernel included with the Xen distribution.

In an SMP guest, each event channel has to be "bound" in all of the preceding three ways. First, the channel must be allocated and bound at both ends. Then, the correct VCPU for handling the events should be bound to the event channel. Finally, the handler function should be bound to the channel, and then the channel unmasked so that event delivery can proceed.

7.6 Operations on Bound Channels

The most obvious thing that can be done to a bound channel is signaling the occurrence of the event along it. This was shown briefly in the example in the last chapter. Signaling an event is very simple. Because event channels are connection-oriented, all that is required is the port number of this end of the channel.

The use of this we saw earlier—to signal the console event channel—and looks something like Listing 7.8. The control structure for the operation only has a single parameter: the port to be signaled.

Listing 7.8: Signaling the console's event channel

```
struct evtchn_send event;
event.port = start->domU.console.evtchn;
HYPERVISOR_event_channel_op(EVTCHNOP_send, &event);
```

Apart from sending a notification along a channel, the only remaining operation that is meaningful to a bound port is closing it. This invalidates the channel, and prevents future events being sent along it. It also recycles the port. The last part is important because you only have eight times the machine word length

event ports, giving a maximum of 256 or 512 channels that can be bound at any one time.

The operation for closing a port (evtchn_close_t) is exactly the same as that for sending a notification to it. It has a single field containing the port number. The command given to the hypercall is EVTCHNOP_close. Any attempt to use the channel from either end will fail after this has been called.

7.7 Getting a Channel's Status

It is sometimes useful to retrieve the status of a channel. This is done by using the EVTCHNOP_status command with the event channel hypercall. This takes a fairly complicated structure as the operation, with two input parameters and a number of output parameters. This structure is shown in Listing 7.9.

Listing 7.9: Event channel status operation

```
1  struct evtchn_status {
2      /* IN parameters */
3      domid_t  dom;
4      evtchn_port_t port;
5      /* OUT parameters */
6      uint32_t status;
7      uint32_t vcpu;
8      union {
9          struct {
10             domid_t dom;
11         } unbound;
12         struct {
13             domid_t dom;
14             evtchn_port_t port;
15         } interdomain;
16         uint32_t pirq;
17         uint32_t virq;
18     } u;
19 };
20 typedef struct evtchn_status evtchn_status_t;
```

The two "in" parameters uniquely identify an endpoint for an event channel as a $(domain, port)$ pair. This pair must indicate a port that has been bound to a channel. If it does not, the hypercall returns an error. The remainder of the structure is filled in by the hypervisor. As usual, unprivileged domains may only set the domain to DOMID_SELF.

Table 7.1: Event channel status values

Constant	Channel Status
EVTCHNSTAT_closed	Not currently in use
EVTCHNSTAT_unbound	Waiting interdomain connection
EVTCHNSTAT_interdomain	Connected to remote domain
EVTCHNSTAT_pirq	Bound to a physical IRQ
EVTCHNSTAT_virq	Bound to a virtual IRQ
EVTCHNSTAT_ipi	Bound for intradomain use

There are two fields that are always filled in for any channel. The first is the status, which must be a value from Table 7.1, indicating the current status of the connection. The other is the VCPU, which is designated to receive events sent along this channel.

The final field is a union, with different values set depending on the value of the status. For an unbound channel, the domain, which is allowed to bind to the remote end, is returned. A channel bound for interdomain communication gives the $(domain, port)$ pair that uniquely identifies the remote end of the connection.

Event channels bound to IRQs, real or virtual have the IRQ number returned. The final category, intradomain event channels (virtual IPIs), return no additional information. The VCPU returned by this hypercall is the one that was specified when the channel was configured.

7.8 Masking Events

When an event handler is called, event delivery to the handling virtual CPU is automatically disabled. Sometimes, it is necessary (or, at least, useful) for events to be masked while performing other operations. Masking events can be performed in either the global scope at a per-event granularity, or on a per-VCPU basis, for all events.

When delivering an event, the hypervisor goes through the steps shown in Figure 7.1. There are three places where it can give up on delivery:

1. If the pending bit is set for the channel. This means that an event is already waiting, so this one can't be handled yet.

2. The event channel is masked. There are a number of reasons why this could happen, but it is typically used to protect the non-reentrant parts of a kernel.

3. The VCPU that would handle the event has elected not to receive events. This is often caused by an event handler in process of running.

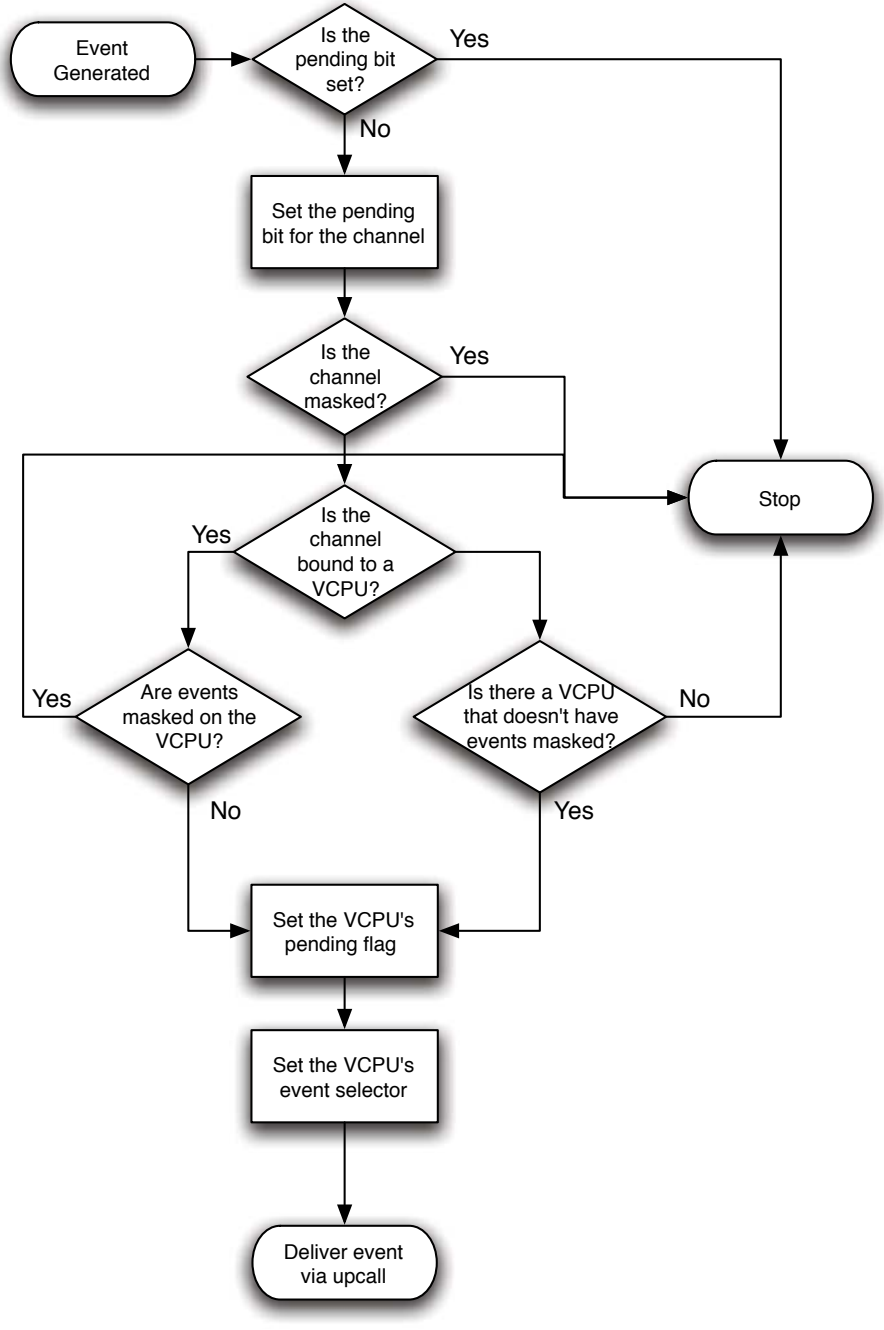

Figure 7.1: The process of delivering an event, from the Hypervisor's perspective

After an event is masked, it is still possible to poll for events by checking the corresponding "pending" bit. This is typically done before exiting the event handler, because other events may have been enqueued while delivery was masked by the handler. It is also very useful for busy event channels. The upcall mechanism is not particularly cheap, in terms of entry and exit costs. If a given event channel is being particularly busy, better performance can be achieved by masking it and polling for events on the channel periodically. This can either be detected at runtime, or implemented at design time for a channel that is expected to be particularly busy.

Both the masking and pending bit fields are found in the shared info page. The hypervisor only ever performs a $0 \rightarrow 1$ transition on the bits in the pending field and does not touch the masking field. The guest may toggle bits in the masked bit field as it wants, and should clear the pending flag when an event has been processed. Note that the hypervisor only tells the guest that "one or more" events have been signaled on a given channel by setting the pending bit. It is up to the guest to do as much processing as is required. For this reason, it is a good idea to clear the pending bit early on in the event handler. This allows events that are delivered while the handler is running to be enqueued. The kernel can then check for pending events on that channel that were raised while the handler was running before unmasking the channel.

7.9 Events and Scheduling

Delivery of events is closely tied to virtual machine scheduling. There are currently four scheduling operations available, two of which are directly connected to event delivery. All are used via the same hypercall. As is conventional, the hypercall takes two arguments: a command and an operation.

The simplest scheduling operation is yield, the equivalent of `sleep(0)` for a UNIX process (or `sched_yield()` for a POSIX thread). It tells the hypervisor that the domain doesn't want to use the rest of its quantum. This operation is issued as follows:

```
HYPERVISOR_sched_op(SCHEDOP_yield, NULL);
```

The operation here is ignored, and so `NULL` is usually passed. When this operation is used, the hypervisor reschedules the domain later. A slightly stronger version of this is the block operation:

```
HYPERVISOR_sched_op(SCHEDOP_block, NULL);
```

This has the same immediate effect as the yield operation: The domain is de-scheduled. It will not be rescheduled, however, until an event is delivered. If all processes in a guest are in blocking states, this allows the guest to wait until

it has something to do before running. The timer VIRQ is delivered via an event channel, so a domain calling this is awakened for its next timer event.

The remaining two scheduling operations are a little more complicated; they both take an argument. The shutdown operation takes an operation with a single field, indicating the reason for the shutdown. Listing 7.10 shows how a guest would perform a clean shutdown.

Listing 7.10: Cleanly shutting down a guest

```
1 sched_shutdown_t op;
2 op.reason = SHUTDOWN_poweroff;
3 HYPERVISOR_sched_op(SCHEDOP_shutdown, &op);
```

Other valid reasons are SHUTDOWN_reboot, if the domain is expecting to be rebooted after shutting down; SHUTDOWN_suspend, if it has disconnected devices and prepared the kernel for suspension; or SHUTDOWN_crash, if the guest has crashed.

This leaves one scheduling operation. Polling is similar to blocking, but is much finergrained. This operation can only be executed when the guest has delivery of all events disabled. The operation used with this command is shown in Listing 7.11.

Listing 7.11: Event polling control structure [from: xen/include/public/sched.h]

```
79 struct sched_poll {
80     XEN_GUEST_HANDLE(evtchn_port_t) ports;
81     unsigned int nr_ports;
82     uint64_t timeout;
83 };
84 typedef struct sched_poll sched_poll_t;
```

The first two fields of this define an array of ports that are being monitored. The timeout field specifies a wall clock time, in nanoseconds since the UNIX epoch. If this time is reached the hypercall returns a nonzero value. Otherwise, it blocks until one of the event channels in the array is signaled. Because event delivery is masked, the event does not produce an upcall, so the caller must determine which event has been delivered and handle it.

7.10 Exercise: A Full Console Driver

In the last chapter, we created a basic console driver. This mapped the memory page used by the console, and allowed some simple output. This example extends the basic driver to trigger events when there is available data and handles it.

All Xen events are received by a single handler. We need to set up a way of dispatching events to a given handler when they are received. We do this by creating a simple handler that dispatches events to their correct handlers. This will store a vector of possible events, and trigger the correct one when required.

The problem comes from the fact that Xen events are delivered completely asynchronously; they can occur at any point in execution, including in the middle of execution of a userspace process. It is therefore necessary to save the current state before proceeding. When entering the event handler callback, the hypervisor masks all events. It is then up to the event handler to reenable them.

When exiting an interrupt handler on x86, it is common to use the **IRET** instruction. This restores control to the process that was interrupted, and re-enables interrupts atomically. This is fine for returning from an interrupt handler; but when returning from an event handler, it is not completely useful, because events are different. Events are an entirely software construct, so the **IRET** instruction has no way of knowing how to enable them. There are two possible solutions to this:

- Provide an **IRET** hypercall
- Don't do it atomically, and pick up the pieces if it all goes wrong

Both of these are used by Xen. The **IRET** hypercall is simple to use, but is quite expensive, because it involves context switching to the hypervisor and back. Most of the time, it turns out that atomicity is not required, because no new events interrupt the old ones. When it is, it's still less expensive to detect and fix than issuing a hypercall every time. Listing 7.12 shows a safe entry point.

Listing 7.12: Event entry point [from: extras/mini-os/arch/x86/x86_32.S]

```
119 ENTRY(hypervisor_callback)
120         pushl %eax
121         SAVE_ALL
122         movl EIP(%esp),%eax
123         cmpl $scrit,%eax
124         jb   11f
125         cmpl $ecrit,%eax
126         jb   critical_region_fixup
127 11:     push %esp
128         call do_hypervisor_callback
129         add  $4,%esp
130         movl HYPERVISOR_shared_info,%esi
131         xorl %eax,%eax
132         movb CS(%esp),%cl
133         test $2,%cl          # slow return to ring 2 or 3
134         jne  safesti
```

```
135 safesti:movb $0,1(%esi)     # reenable event callbacks
136 scrit:  /**** START OF CRITICAL REGION ****/
137         testb $0xFF,(%esi)
138         jnz  14f              # process more events if
                necessary...
139         RESTORE_ALL
140 14:     movb $1,1(%esi)
141         jmp  11b
142 ecrit:  /**** END OF CRITICAL REGION ****/
```

The real event handler is in the do_hypervisor_callback function, defined elsewhere. The first part of this reads the origin address, and checks whether it falls between the scrit : and ecrit : labels. If it is, this means the current event is interrupting the event handler, rather than real code. This is a problem because we have restored the values of some registers from where we stored them (on the stack), but not all of them. If this happens, we have a problem; there will be one complete copy of registers on the stack, and some rubbish. We need to merge the two stack-based copies together.

Listing 7.13 shows how this is done, if detected. The critical fix-up table referenced here is a simple table structure that allows determination of how many have been successfully restored. This then copies the values lower down the stack and sets the stack pointer to the correct value. When the two parts of the stack-copy of the register file are in the same place, the RESTORE_ALL macro (which just pops them all off the stack) can restore them as if the interruption had not happened. After this happens, it jumps back into the event handler, which then keeps trying to process events until there are none left.

The overall flow of this bit of code is to enter the event handler and push the contents of all registers onto the stack. Next, it checks if you came from the critical section, which is the part between reenabling event delivery and returning. If this is the case, the stack frame is remangled so that it contains the same thing it would have contained if the interruption had not occurred. After this, it loops, repeatedly processing pending events, until it runs out, at which point it reenables event delivery, restores the register file, and returns.

The contents of the stack pointer is pushed just prior to calling the real handler. This points to the previous stack frame, which contains the contents of all of the registers from prior to the event. This allows the event handler to have access to the processor state, and optionally tweak things if required.

Listing 7.13: Critical region interruption fix [from: extras/mini-os/arch/x86/x86_32.S]

```
150 critical_region_fixup:
151         addl $critical_fixup_table-scrit,%eax
152         movzbl (%eax),%eax    # %eax contains num bytes popped
153         mov  %esp,%esi
```

```
154         add  %eax,%esi          # %esi points at end of src
                region
155         mov  %esp,%edi
156         add  $0x34,%edi         # %edi points at end of dst
                region
157         mov  %eax,%ecx
158         shr  $2,%ecx            # convert words to bytes
159         je   16f                # skip loop if nothing to copy
160 15:     subl $4,%esi            # pre-decrementing copy loop
161         subl $4,%edi
162         movl (%esi),%eax
163         movl %eax,(%edi)
164         loop 15b
165 16:     movl %edi,%esp          # final %edi is top of merged
    stack
166         jmp  11b
```

This bit of event handler code is highly platform-specific, and needs to be rewritten for other architectures. It gives some abstraction, however, and our "real" event handler can now be in do_hypervisor_callback , which can be defined as some C code. This, quite messy, function is shown in Listing 7.14, and is from the `event.c` file in this chapter's sample code.

Listing 7.14: Event callback function [from: examples/chapter7/event.c]

```
67 /* Dispatch events to the correct handlers */
68 void do_hypervisor_callback(struct pt_regs *regs)
69 {
70     unsigned int pending_selector;
71     unsigned int next_event_offset;
72     vcpu_info_t *vcpu = &shared_info.vcpu_info[0];
73     /* Make sure we don't lose the edge on new events... */
74     vcpu->evtchn_upcall_pending = 0;
75     /* Set the pending selector to 0 and get the old value
           atomically */
76     pending_selector = xchg(&vcpu->evtchn_pending_sel, 0);
77     while(pending_selector != 0)
78     {
79         /* Get the first bit of the selector and clear it */
80         next_event_offset = first_bit(pending_selector);
81         pending_selector &= ~(1 << next_event_offset);
82         unsigned int event;
83
84         /* While there are events pending on unmasked channels
               */
85         while(( event =
86         (shared_info.evtchn_pending[pending_selector]
```

```
87              &
88             ~shared_info.evtchn_mask[pending_selector]))
89              != 0)
90             {
91                 /* Find the first waiting event */
92                 unsigned int event_offset = first_bit(event);
93
94                 /* Combine the two offsets to get the port */
95                 evtchn_port_t port = (pending_selector << 5) +
                       event_offset;
96                 /* Handle the event */
97                 handlers[port](port, regs);
98                 /* Clear the pending flag */
99                 CLEAR_BIT(shared_info.evtchn_pending[0],
                       event_offset);
100            }
101        }
102    }
```

The first thing this does is clear the pending upcall flag. If another event arrives, this flag is set again and the hypervisor delivers another upcall later. Clearing this later could, potentially, result in some events being deferred until a subsequent event is delivered.

The pending event selector is then cleared, and a copy of it retained. This has one bit for each word in the event bitfields. The outer loop scans over this selector, once for each set bit, and looks in the corresponding word in the event bitmap. Note that this is a fairly x86-specific optimization; on platforms that don't have bitfield-manipulation instructions, the extra shifts and tests are likely to be much slower than simply testing each word against zero.

The inner loop runs while there are events that are pending and not masked. For each one, it calls the corresponding handler, and then clears the pending bit. When there are no pending, unmasked, events left, the function returns, dropping back to the assembly code, which then reenables interrupts.

Of course, before this can happen, the event handlers need to be set up. Listing 7.15 shows the two other functions required for this. The first begins by registering the two hypervisor upcall entry points. The first is used for event delivery, and the second is used when something goes horribly wrong. It then sets up a default handler (which does nothing) for all events and masks the relevant channel. Finally, it enables upcalls. In most cases, you then check for missed events after doing this. We don't have to, because we know that we have just masked all event channels, so an upcall would not call any handlers. The second function assigns a handler function for the port and unmasks it.

We use the second function in our new console initializer, in Listing 7.16. This is remarkably similar to the last one, but registers a handler for an event when

Listing 7.15: Setting up event handlers [from: examples/chapter7/event.c]

```
21 static evtchn_handler_t handlers[NUM_CHANNELS];
22
23 void EVT_IGN(evtchn_port_t port, struct pt_regs * regs) {};
24
25 /* Initialise the event handlers */
26 void init_events(void)
27 {
28     /* Set the event delivery callbacks */
29     HYPERVISOR_set_callbacks(
30         FLAT_KERNEL_CS, (unsigned long)hypervisor_callback,
31         FLAT_KERNEL_CS, (unsigned long)failsafe_callback);
32     /* Set all handlers to ignore, and mask them */
33     for(unsigned int i=0 ; i<NUM_CHANNELS ; i++)
34     {
35         handlers[i] = EVT_IGN;
36         SET_BIT(i,shared_info.evtchn_mask[0]);
37     }
38     /* Allow upcalls. */
39     shared_info.vcpu_info[0].evtchn_upcall_mask = 0;
40 }
```

input data is ready. The handler for this event is shown in Listing 7.17. It's very simple, and just checks if there is actually any data waiting and copies it to the output if there is.

Setting up event handling is almost completed. All we need to do now is configure our kernel to call the relevant initialization functions. The body of the kernel is shown in Listing 7.18. The first thing this does is map the shared info page over the space left for it in the kernel image. It also sets a pointer to this, which is used by the assembly bootstrap to locate the upcall masking flag and clear it.

Next, it calls the event and console initialization routines that we have just described, and performs the obligatory output of "Hello world!" to prove that it all worked. Finally, it loops. This infinite loop is slightly different from previous ones, in that it tells the hypervisor not to schedule the guest at all until an event is received. The event is then processed via the upcall, and control is returned to the loop, which immediately sleeps again. This model can be used to create purely event-driven kernels.

If all of this works, our kernel should run, accepting input and echoing it to the screen:

Listing 7.16: The new console initializer [from: examples/chapter7/console.c]

```
24 /* Initialise the console */
25 int console_init(start_info_t * start)
26 {
27     console = (struct xencons_interface*)
28         ((machine_to_phys_mapping[start->console.domU.mfn] <<
              12)
29          +
30         ((unsigned long)&_text));
31     console_evt = start->console.domU.evtchn;
32     /* Set up the event channel */
33     register_event(console_evt, handle_input);
34     return 0;
35 }
```

Listing 7.17: The console event handler [from: examples/chapter7/console.c]

```
10 /* Event received on console event channel */
11 void handle_input(evtchn_port_t port, struct pt_regs * regs)
12 {
13     XENCONS_RING_IDX cons = console->in_cons;
14     XENCONS_RING_IDX prod = console->in_prod;
15     int length = prod - cons;
16     if(length > 0)
17     {
18         char buffer[10];
19         console_read(buffer, ++length);
20         console_write(buffer);
21     }
22 }
```

Listing 7.18: The event-handling kernel body [from: examples/chapter7/kernel.c]

```
10 shared_info_t *HYPERVISOR_shared_info;
11
12 /* Main kernel entry point, called by trampoline */
13 void start_kernel(start_info_t * start_info)
14 {
15     /* Map the shared info page */
16     HYPERVISOR_update_va_mapping((unsigned long) &shared_info,
17             __pte(start_info->shared_info | 7),
18             UVMF_INVLPG);
19     /* Set the pointer used in the bootstrap for reenabling
20      * event delivery after an upcall */
21     HYPERVISOR_shared_info = &shared_info;
22     /* Set up and unmask events */
23     init_events();
24     /* Initialise the console */
25     console_init(start_info);
26     /* Write a message to check that it worked */
27     console_write("Hello world!\r\n");
28     /* Loop, handling events */
29     while(1)
30     {
31         HYPERVISOR_sched_op(SCHEDOP_block,0);
32     }
33 }
```

```
# xm create -c domain_config
Using config file "./domain_config".
Started domain Simplest_Kernel
Hello world!
Does this work?  Yes, seems to... ^]
# xm list
Name                ID   Mem VCPUs      State   Time(s)
Domain-0             0   250     1     r-----   1376.4
Simplest_Kernel    181    32     1     -b----      0.0
```

Note the output from `xm list`, specifically the time column. Because we are blocking, this only increments (by a few milliseconds) when a key is pressed, generating an event. This is a contrast with our earlier attempts, which spent a lot of time spinning needlessly.

Chapter 8

Looking through the XenStore

The XenStore is a storage system shared between Xen guests. It is a simple hierarchical storage system, maintained by Domain 0 and accessed via a shared memory page and an event channel. Although the XenStore is fairly central to the operation of a Xen system, there are no hypercalls associated with it. The start info page contains the address of the shared memory page used to communicate with the store. A guest maps this page and then all further communication happens via the ring buffers in this page.

This chapter will explore the contents of the XenStore and the ways of interacting with it from userspace in existing systems and from a newly ported kernel. Unlike many other parts of Xen, the XenStore has a well-defined interface for userspace programs to use, intended to reduce the dependency of the tools on features of the host operating system.

8.1 The XenStore Interface

The hypervisor itself is not aware of the existence of the store. It is maintained by a dæmon running in Domain 0, and accessed much like any other device driver. The interfaces to device drivers will be covered in more detail in the next section.

The basic interface to the store consists of two ring buffers, one for each direction. Requests to update the store or for information about the current contents are placed into one ring. Responses and asynchronous notifications of changes are inserted into the other ring. The first ring is written to by domU guests and read from by dom0; the other ring is written to by dom0 and read from by domU.

The XenStore is composed of directories, which can contain other directories

or keys. Each key has an associated value. The store could also be thought of as a nested associative array, or dictionary. The structure is very similar to a filesystem. The use is somewhat different, however. It is not intended that the store should be used for storing or transferring large amounts of data; indeed, the interface makes it quite difficult to use it in this way. It is mainly used as an extensible method of transmitting small amounts of information between domains. For example, the location of virtual devices is exposed via the XenStore.

The store is also used to provide information about running domains in a fairly easily readable format. This can then be accessed by the administrative tools to provide information to an administrator, and to give a persistent place for them to store their own information.

Unlike most filesystems, the XenStore supports a transactional model for I/O. Groups of requests can be bundled into a transaction, assuring that they will complete atomically. This allows consistent views of the store (or some subtree) to be easily created.

XenBus

You may hear the term XenBus thrown around in various contexts. There is some confusion about exactly what this means, caused by the fact that the same term is used to describe two things. In the Xen port of Linux, the term is used to describe the interface to the XenStore. In a more general case, it is used to describe a protocol for connecting to device drivers that is built on top of the XenStore.

8.2 Navigating the XenStore

The Xen distribution includes a number of command-line tools for inspecting and manipulating the store. These are the `xenstore-*` family, and can be used from Domain 0. To get a full list of the contents of the XenStore, use the `xenstore-ls` command.

```
# xenstore-ls
tool = ""
...
vm = ""
 00000000-0000-0000-0000-000000000000 = ""
...
local = ""
...
```

The three top-level entities in the store are tool, vm, and local. The tool hierarchy is used for tools to store information. This part of the hierarchy exists to allow tools to have a uniform storage and communication mechanism, isolated from the underlying filesystem of Domain 0.

The /vm tree contains an entry for each virtual machine running. Each is identified by a globally unique ID. The information in this tree is fairly static, and should not change much over the lifetime of the guest. The indexing by means of a UUID allows this tree to be duplicated after migration. Domain 5 on one machine may be migrated to domain 12 on another, but their UUID will remain constant. By specifying an exact path to `xenstore-ls`, you can look at a single VM's entry. This shows the output for an instance of the Xen Mini OS:

```
# xenstore-ls /vm/1d6af2c8-edf6-fc8c-c659-222ebbf3feea
image = "(linux (kernel /root/xen-src/extras/mini-os/mini-os.elf))"
 ostype = "linux"
 kernel = "/root/xen-src/extras/mini-os/mini-os.elf"
 cmdline = ""
 ramdisk = ""
shadow_memory = "0"
uuid = "1d6af2c8-edf6-fc8c-c659-222ebbf3feea"
on_reboot = "restart"
start_time = "1176124400.97"
on_poweroff = "destroy"
name = "Mini-OS"
xend = ""
 restart_count = "0"
vcpus = "1"
vcpu_avail = "1"
memory = "32"
on_crash = "destroy"
maxmem = "32"
```

Many of the settings here are provided by the domain configuration file used to create the domain. The ostype field is slightly misleading. It refers to the domain builder used to create this domain. At present, two domain builders come with Xen: Linux and HVM. Linux is a fairly generic ELF loader with support for loadable modules, whereas HVM is designed to boot unmodified operating systems. Adding a new domain builder requires patching Xen. Both Plan 9 and Minix do this currently, in order to support booting an a.out format kernel image.

The `vcpus` and `vcpu_avail` fields indicate the number of virtual CPUs that the domain has access to. The first value contains the number that are allocated to the domain in total, and the second the number that are not disabled. Some

of the information in the /vm tree is duplicated in the /local/domain tree, which indexes virtual machines by domain. This tree also includes runtime information, such as the configuration of connected devices. The same Mini OS domain has this information in the /local/domain part of the tree:

```
# xenstore-ls /local/domain/1
console = ""
 ring-ref = "5400"
 port = "2"
 limit = "1048576"
 tty = "/dev/ttyp1"
name = "Mini-OS"
vm = "/vm/1d6af2c8-edf6-fc8c-c659-222ebbf3feea"
domid = "1"
cpu = ""
 0 = ""
  availability = "online"
memory = ""
 target = "32768"
store = ""
 ring-ref = "5401"
 port = "1"
```

Mini OS, being simpler than most guests, only uses two devices: the console and the XenStore. As mentioned earlier, the locations of these are both passed in via the shared info pages at boot time. The ring-ref and port entries in the store should match the grant references and event channel numbers from the start info page. The console subtree also includes information relating to the implementation of the back end, such as the amount of data that will be buffered and the device used by the console. The buffer is not the ring buffer used for I/O, but something similar to the scroll-back buffer on a virtual terminal. In this example, 1MB of text can be stored here. This is more important on a VM than a physical machine, because the console on a VM is likely not to be connected to a real device for much of the time, and so maintaining history is important.

The /local/domain tree is conceptually similar to the `/proc` hierarchy on a UNIX system, a virtual filesystem containing a directory for every process running on the system. The contents of `/proc` vary widely between UNIX-like systems (Linux, for example, uses it as a substitute for sysctls in many cases, and XNU omits it entirely), but at a minimum it is likely to contain one entry for each valid PID in the system. The local domain hierarchy in the XenStore is arranged similarly, with one entry for each domain in the system. Domains in Xen correspond very closely to processes in UNIX. Each has its own address space, and can have

multiple threads (VCPUs) that may or may not execute concurrently, depending on the nature of the number of physical CPUs in the system.

8.3 The XenStore Device

The XenStore was discussed from a high-level perspective in the previous chapter. Now we will take a look at exactly how the guest interacts with it. The page used by the XenStore is mapped into the guest's address space by the domain builder, and the machine frame number is provided by the start info page.

To use this page, you must first obtain a pointer to it. On x86, and other platforms where the guest is aware of the difference between pseudo-physical and machine addresses, you can do this using the machine frame list. This is placed in the top of the guest's virtual address space, in a region that is read-only to the guest, and identified by the variable machine_to_phys_mapping. This is an array of pseudo-physical frame numbers, indexed by machine frame number. The pseudo-physical frame of the store's page will be:

```
machine_to_phys_mapping[start_info.store_mfn]
```

Remember that this is a frame number, rather than an address. To turn it into an address, you must multiply it by the size of a single page, typically 4K. This is typically done by means of a left-shift, because the page size is always a power of two. The address can then be used as a pointer to a structure of the form shown in Listing 8.1, and defined in the `io/xs_wire.h` public header.

Listing 8.1: XenStore interface structure [from: xen/include/public/io/xs_wire.h]

```
99  #define MASK_XENSTORE_IDX(idx) ((idx) & (XENSTORE_RING_SIZE-1))
100 struct xenstore_domain_interface {
101     char req[XENSTORE_RING_SIZE]; /* Requests to xenstore
            daemon. */
102     char rsp[XENSTORE_RING_SIZE]; /* Replies and async watch
            events. */
103     XENSTORE_RING_IDX req_cons, req_prod;
104     XENSTORE_RING_IDX rsp_cons, rsp_prod;
105 };
```

This structure contains two ring buffers—one for requests and one for responses—and their associated producer and consumer counters. Note that these do not use the ring macros described earlier. This is because they do not fit into the neat one-request-per-response idea that the generic ring macros expect. A request might register a watch, and then have several responses every time the watched node is updated. Worse, the first response may come a long time after

the request is added. Using the generic ring model, we would quickly run out of space in the rings. Instead, requests are enqueued in one ring, and responses in another, with an event (described in more detail in the next chapter) used to signal the arrival of a new response.

Requests sent to the back end are fairly similar conceptually to network packets. The fact that the structure used to represent them is called xsd_sockmsg only helps to reinforce this parallel. The structure is actually the message header, rather than the entire message, and contains four fields, all of which are 32-bit integers, as shown here:

```
struct xsd_sockmsg
{
    uint32_t type;
    uint32_t req_id;
    uint32_t tx_id;
    uint32_t len;
};
```

The first indicates the type of the packet, and must be selected from an enumerated type declared in the same file. The second is a unique identifier. Any responses to this request will have the same req_id value. The tx_id field is used to indicate which transaction a request is part of. If a number of requests have to complete atomically, they can be grouped inside a pair of requests with the XS_TRANSACTION_START and XS_TRANSACTION_END types, respectively, and have the same transaction ID set for all of them. The back end should wait until the end of the transaction, lock any relevant data structures, and then execute the transaction in one go. Finally, the len field indicates the length of the body.

After the header, the body of the message is generally a text string. The XenStore, for simplicity, is based around text strings. If more than one string is required for a given request or response, they are separated by NULLs. For example, if you are writing a value, the path of the key and the associated value will both be provided as strings in the message body. Note that this imposes a practical limitation on the size of XenStore, because the length of a key, value and the message header must all be able to fit in the request ring at once in order to set them. In the current implementation, the rings are 1024 bytes each.

The types of XenStore messages are all prefixed with XS_. The simplest are XS_READ and XS_WRITE, responsible for reading and writing a key, respectively.

When reading a key, the XS_READ command is used in the type field of the message header, and the message body is set to a NULL-terminated string indicating the key. A response is then enqueued in the response ring with the same req_id. If the type of the response is XS_ERROR, the request failed. In this case, the message body will contain the name of the error—for example, *"ENOENT"*—if the path did not exist. Note that a string is returned for an error, rather than a symbolic value.

If the read request worked correctly, the message body of the response will contain the value associated with the key. A write request is handled in a similar way, although the request should contain two NULL-terminated (and, thus, NULL-separated) strings, the first indicating the key path and the second indicating the value. Errors are reported in the same way, and a response of a type other than XS_ERROR indicates success.

Keys paths are written in the same style as UNIX file paths, so `/local/domain/0/name` is used to represent the path to the name of the dom0 guest. Each VM has its own subtree in `/local/domain` represented by its domain number, and all relative paths are assumed to start here. This allows a domain to check for devices without having to be aware of its domain number. The path `device/vbd/0`, for example, always points to the first virtual block device available to this domain. A list of all available device categories can be discovered by enumerating the keys in the `device` hierarchy, and the available devices of each type by enumerating keys in children. For example, the available virtual network interfaces can be discovered by enumerating the children of the `device/vif` key.

Enumerating keys is done using the XS_DIRECTORY message type. This returns a list of NULL-terminated strings indicating the children of the given key. Note, again, that the children of any given key must fit into the response ring in order for a guest to be able to enumerate them. This is not typically a problem, because the size of keys is small, as is their number.

If you are considering storing other information in the XenStore, however, keep these limitations in mind. The XenStore should be used as a mechanism for communicating small amounts of data. It might be tempting to use it as a general purpose storage or communications system, but block devices, virtual interfaces, or shared memory pages are generally better for this kind of use.

8.4 Reading and Writing a Key

The primary use of the XenStore is to store key-value pairs in a location that is easily accessible to running guests and tools. The userspace tools may use the store to contain their own configuration information, or to set information intended to be read by other guests.

There are three ways in which it is possible to access the XenStore. You can use the command-line tools, from a shell in a running system that supports Python and has had the tools ported. This is the best way for accessing the XenStore from shell scripts, or for performing simple administrative tasks that require XenStore interaction. Going to a slightly lower layer, we can use the C API, which communicates with a XenStore device exported by the host operating system. Finally, we can use the kernelspace interface directly. The last of these is most likely to be used on a newly ported guest, because the higher level approaches

cannot be used until the kernel exposes an interface for the tools to use. You will examine how each of these works.

8.4.1 The Userspace Way

The store can be used from the command line relatively easily to read and write keys. The `xenstore-read` and `xenstore-write` commands are used for this purpose.[1] We will try creating, inspecting, and removing a key in the `/example` namespace, in a variety of ways. First, simply using the existing tools, you can run the following commands:

```
# xenstore-write /example ""
# xenstore-write /example/foo bar
# xenstore-list /example
foo
# xenstore-read /example/foo
bar
# xenstore-rm /example/foo
# xenstore-list /example
#
```

Note that, outside example usage, it is generally considered to be a bad idea to create a new key in the root of the XenStore.

The first command creates the dictionary that will be used to store the rest of the example. The next creates a key inside this called `foo`, with the contents "bar." It is worth noting that keys have both values and children. The `/example` key contains the empty string as its value, but it also contains the `foo` key as a child. Convention within the XenStore is that each key should only have either values or children; however, this is not enforced.

The `xenstore-list` command is a nonrecursive version of the `xenstore-ls` command shown earlier. This command only lists the children of a particular key; it does not display their values nor the children of children. This command corresponds a lot more closely with the underlying XenStore commands being issued than the `xenstore-ls`.

We will now take a look at what the command is actually doing when you invoke it. All of the XenStore functions are exported to the userspace tools via `libxenstore`. This library and the tool that uses it are both in the `tools/xenstore` part of the Xen tree. First, we will take a look at the tool.

The `xenstore-*` family of tools are all built from the same source file, which is full of conditional compilation directives. The file is `xenstore_client.c`. A

[1] The command-line examples in this section are run from Domain 0. Depending on your configuration, some may fail when executed from other domains.

lot of the code here is for handling the arguments. The actual code calling the library is very simple. The first thing that needs to happen is for the program to establish a connection to the XenStore:

```
xsh = socket ? xs_daemon_open() : xs_domain_open();
```

The xsh variable is a pointer to an xs_handle structure. This is instantiated using one of two functions, depending on whether the -s option was specified, setting the socket variable.

If a transaction is being used, the tool next needs to create one:

```
xth = xs_transaction_start(xsh);
```

This gives a xs_transaction_t containing the transaction ID. A value of XBT_NULL is returned to indicate an error. The return value for this function should be passed to other XenStore functions to indicate that the associated operation should be regarded as being part of the transaction. Alternatively, the value XBT_NULL can be used to indicate operations that are not part of any transaction.

At the end of the transaction, it is completed using this call:

```
xs_transaction_end(xsh, xth, ret);
```

This ends transaction xth on connection xsh, with ret containing a Boolean value indicating whether the operations within the transaction succeeded. If they failed, the entire transaction should be reverted.

In between the start and end of the transaction, some operations need to be performed. The simplest of these is reading a key:

```
char *val = xs_read(xsh, xth, argv[optind], NULL);
```

The third argument of this function is the name of the key to read. If multiple keys are passed on the command line, the function will loop over them, incrementing the value of optind until it has attempted to read all of the values. The final argument is a pointer to the length of the returned string. If this is NULL, it will be ignored.

The returned string will be malloc()'d by the callee, and must be freed by the caller. The same general structure is used for all interactions with the XenStore. When we tried to list the children of a key, the command used the following API call:

```
char **list = xs_directory(xsh, xth, argv[optind], &num);
```

This is almost identical to the read function. Here, an array of strings is returned, rather than a single string, and the final argument indicates the length of the array (the number of elements) rather than the length of a single string.

The function for writing into the XenStore uses a slightly different structure:

```
xs_write(xsh, xth, argv[optind], argv[optind + 1],
                        strlen(argv[optind + 1]));
```

This returns a Boolean value indicating an error (following the C convention of zero indicating success). The first two arguments, as always, are the XenStore connection and the transaction ID. The next are the key and value to write, followed by the length of the value. When creating our `/example` key into which the remainder of the example was inserted, the call looked something like this:

```
xs_write(xsh, xth, "/example", "", 1);
```

Note that the length includes the terminating NULL. The smallest value that can be stored is a single zero.

8.4.2 From the Kernel

A guest kernel can interact with the XenStore with a similar degree of control. The XenStore, like most other devices, is interacted with via a shared memory page and an event channel. In implementation, it is similar to the console device. Unlike other devices, which retrieve their configuration information from the XenStore, the store has its page mapped into the guest's address space and the event channel connected on system boot.

The two pieces of information required to begin using the XenStore are found in the start info page, in the store_mfn and store_evtchn fields. The first of these gives the machine frame number of the shared memory page containing the XenStore ring buffer. This must be converted to a virtual address before it can be used. The other is the event channel. A handler should be configured for this, as discussed in the last chapter.

The XenStore is very similar, in terms of interface, to the console device. Both are mapped into the new domain's address space by the domain builder, and have their event channels assigned at boot time. Both have two rings, one for requests and the other for responses, and producer/consumer counters for each in the shared ring. Both mainly deal with text.

Setting up the XenStore device, as with the console, is simply a matter of getting the pseudo-physical address of the shared page and keeping this as a pointer. After that, an event handler should be set up for retrieving asynchronous responses, both from requests and from watches. Listing 8.2 shows the basic initialization required for the store.

The biggest difference between the XenStore interface and most others is the way in which data is sent between the front and back ends. The console is fairly unique in that it doesn't have discrete requests and responses, instead providing a stream interface. Most other drivers, however, have fixed-length messages. The XenStore is closer to a packet-based interface. The XenStore message structure, shown in Listing 8.3, only represents the "header" for the message, rather than the entire message. The body is then a plain-text string.

Listing 8.2: Setting up the XenStore [from: examples/chapter8/xenstore.c]

```
21 /* Initialise the XenStore */
22 int xenstore_init(start_info_t * start)
23 {
24     xenstore = (struct xenstore_domain_interface*)
25         ((machine_to_phys_mapping[start->store_mfn] << 12)
26          +
27         ((unsigned long)&_text));
28     xenstore_evt = start->store_evtchn;
29     /* TODO: Set up the event channel */
30
31     return 0;
32 }
```

Listing 8.3: XenStore message header [from: xen/include/public/io/xs_wire.h]

```
80 struct xsd_sockmsg
81 {
82     uint32_t type;  /* XS_??? */
83     uint32_t req_id;/* Request identifier, echoed in daemon's
           response.  */
84     uint32_t tx_id; /* Transaction id (0 if not related to a
           transaction). */
85     uint32_t len;   /* Length of data following this. */
86
87     /* Generally followed by nul-terminated string(s). */
88 };
```

For now, we will write a purely synchronous implementation of the store. Rather than putting messages into the request queue and then processing the responses in the callback, we will use a scheduler operation to poll for the response and wait until it is found. This stops our implementation from being re-entrant, so only a single thread in the kernel may access it at once without locking, but makes it a lot easier to see the execution flow.

Most of our interactions with the store require writing a message into the buffer and signaling the back end. Because this is going to be used a few times, we will put it in a function that can be called from elsewhere. This function, shown in Listing 8.4, is quite similar to the code for writing to the console. The main difference is that we don't support writing messages bigger than the size of the buffer. Because the console was stream based, we could write a part of the message and then tell the back end to handle it and then write the rest. We can't

do this with the XenStore, because requests should be processed as a complete message. If the message is too big for the buffer, we just return.

Listing 8.4: Writing a message to the XenStore back end [from: examples/chapter8/xenstore.c]

```
34 /* Write a request to the back end */
35 int xenstore_write_request(char * message, int length)
36 {
37     /* Check that the message will fit */
38     if(length > XENSTORE_RING_SIZE)
39     {
40         return -1;
41     }
42
43     int i;
44     for(i=xenstore->req_prod ; length > 0 ; i++,length--)
45     {
46         /* Wait for the back end to clear enough space in the
               buffer */
47         XENSTORE_RING_IDX data;
48         do
49         {
50             data = i - xenstore->req_cons;
51             mb();
52         } while (data >= sizeof(xenstore->req));
53         /* Copy the byte */
54         int ring_index = MASK_XENSTORE_IDX(i);
55         xenstore->req[ring_index] = *message;
56         message++;
57     }
58     /* Ensure that the data really is in the ring before
           continuing */
59     wmb();
60     xenstore->req_prod = i;
61     return 0;
62 }
```

The other difference is that we need to explicitly state the length of our message. For console output, we could simply use a terminating NULL to detect the end of the message. XenStore messages, however, use these as separators. When writing to the store, for example, both the key and value will be passed as NULL-terminated strings. If we stopped sending after reaching the zero byte, we would send the key but not the value.

We will also define a function, shown in Listing 8.5 for reading a response from the store. This reads a fixed size message from the store into a preprepared

buffer. We can't just use the data from the ring, unfortunately, because when it wraps around the end of the buffer we would have to track the discontinuity. In principle, we could only copy noncontiguous messages, but for simplicity we will always copy.

Listing 8.5: Reading a response from the XenStore [from: examples/chapter8/xenstore.c]

```
64 /* Read a response from the response ring */
65 int xenstore_read_response(char * message, int length)
66 {
67     int i;
68     for(i=xenstore->rsp_cons ; length > 0 ; i++,length--)
69     {
70         /* Wait for the back end put data in the buffer */
71         XENSTORE_RING_IDX data;
72         do
73         {
74
75             data = xenstore->rsp_prod - i;
76             mb();
77         } while (data == 0);
78         /* Copy the byte */
79         int ring_index = MASK_XENSTORE_IDX(i);
80         *message = xenstore->rsp[ring_index];
81         message++;
82     }
83     xenstore->rsp_cons = i;
84     return 0;
85 }
```

We also define the variable and macros shown in Listing 8.6. The req_id variable contains the next request ID to use. Every time we issue a new request, we increment this counter. The macros are used to notify the back end via an event channel, and to ignore part of a response. The latter is used when a response includes some extra information that isn't required by the requester. For now, we use it for ignoring errors, although a full implementation should handle them properly.

This example first required writing a key, so we'll implement that first. The basic process for doing this can be viewed as follows:

1. Prepare the message header.
2. Send the header.
3. Send the $(key, value)$ pair.
4. Signal the event channel.

Listing 8.6: Macros from the XenStore driver [from: examples/chapter8/xenstore.c]

```
87 /* Current request ID */
88 static int req_id = 0;
89
90 #define NOTIFY() \
91     do {\
92         struct evtchn_send event;\
93         event.port = xenstore_evt;\
94         HYPERVISOR_event_channel_op(EVTCHNOP_send, &event);\
95     } while(0)
96
97 #define IGNORE(n) \
98     do {\
99         char buffer[XENSTORE_RING_SIZE];\
100        xenstore_read_response(buffer, n);\
101    } while(0)
```

5. Read the response.

The function shown in Listing 8.7 does exactly this. The three components of the message—the header, key, and value—are written in turn, and then a response is read. This basic implementation doesn't check the return value for an error, it just ignores it.

Listing 8.7: Writing a key in the XenStore [from: examples/chapter8/xenstore.c]

```
103 /* Write a key/value pair to the XenStore */
104 int xenstore_write(char * key, char * value)
105 {
106     int key_length = strlen(key);
107     int value_length = strlen(value);
108     struct xsd_sockmsg msg;
109     msg.type = XS_WRITE;
110     msg.req_id = req_id;
111     msg.tx_id = 0;
112     msg.len = 2 + key_length + value_length;
113     /* Write the message */
114     xenstore_write_request((char*)&msg, sizeof(msg));
115     xenstore_write_request(key, key_length + 1);
116     xenstore_write_request(value, value_length + 1);
117     /* Notify the back end */
118     NOTIFY();
119     xenstore_read_response((char*)&msg, sizeof(msg));
120     IGNORE(msg.len);
```

```
121     if(msg.req_id != req_id++)
122     {
123         return -1;
124     }
125     return 0;
126 }
```

After we can write a key, the next thing to do is try reading it back. This procedure here is very similar. We first prepare the header. The structure of a read message is slightly simpler than a write; it only has the header and key, not a value. The response uses the same packet header as the request, and is followed by the value.

Listing 8.8 shows how we read a value. First we write the message header with the XS_READ type and then the key we want to read. Note that keys are in fact key paths, separated by the / character, although the interface makes no distinction between local keys and full or relative paths.

These messages both return errors in the same way. Rather than returning the expected string, they return a string containing the error value, such as "EINVAL" for an invalid key or "EACCES" for a permissions error. This is not a problem when writing a key,[2] because you do not expect a string return value and so anything after the header is going to be an error code. For a read operation, where a return value is expected, it may cause problems. As long as you avoid writing error codes as keys or values in the XenStore, it should be possible to distinguish the two. By convention, XenStore entries are all lowercase, which makes them easy to distinguish from error codes, although there is nothing stopping you from writing uppercase strings to the store.

Listing 8.8: Reading a value from the XenStore [from: examples/chapter8/xenstore.c]

```
128 /* Read a value from the store */
129 int xenstore_read(char * key, char * value, int value_length)
130 {
131     int key_length = strlen(key);
132     struct xsd_sockmsg msg;
133     msg.type = XS_READ;
134     msg.req_id = req_id;
135     msg.tx_id = 0;
136     msg.len = 1 + key_length;
137     /* Write the message */
138     xenstore_write_request((char*)&msg, sizeof(msg));
139     xenstore_write_request(key, key_length + 1);
140     /* Notify the back end */
141     NOTIFY();
142     xenstore_read_response((char*)&msg, sizeof(msg));
```

[2] And isn't a problem for this implementation, which pretends errors never happen.

```
143     if(msg.req_id != req_id++)
144     {
145         IGNORE(msg.len);
146         return -1;
147     }
148     /* If we have enough space in the buffer */
149     if(value_length >= msg.len)
150     {
151         xenstore_read_response(value, msg.len);
152         return 0;
153     }
154     /* Truncate */
155     xenstore_read_response(value, value_length);
156     IGNORE(msg.len - value_length);
157     return -2;
158 }
```

The other operation we will add to the simple XenStore driver is a xenstore_ls function, not listed here. The request part of this looks exactly like the read function, with a type of XS_DIRECTORY instead of XS_READ. Beyond that, the implementation is exactly the same. The only remaining difference is that the response is now a NULL-separated list of strings, rather than a single string. To help the caller split them up, the function is modified so that it returns the total length of the array, as provided in msg.len.

To ensure all of this is working correctly, we add one final function to test the store. This first attempts to retrieve the domain's name from the `name` key. It then writes a value of "foo" to `example` and tries to read it back. Finally, it lists the keys in the `console` dictionary.

Listing 8.9 shows the full implementation of the testing function. This and xenstore_init () are called from the kernel's main function. Note the stack-allocated buffer here. This is fairly reasonable in a userspace program (security considerations of reading data to the stack aside), but could cause problems for our tiny kernel, which has a statically allocated 8KB stack. It should be okay here, because there is only one function above it on the stack, and only one below. Deep event handlers could be a problem, but there are none here. In general, however, it is not a good practice.

Listing 8.9: Testing the XenStore [from: examples/chapter8/xenstore.c]

```
191 /* Test the XenStore driver */
192 void xenstore_test()
193 {
194     char buffer[1024];
195     buffer[1023] = '\0';
196     console_write("\n\r");
```

```
197     /* Get the name of the running VM */
198     xenstore_read("name", buffer, 1023);
199     console_write("VM name: ");
200     console_write(buffer);
201     console_write("\n\r");
202     /* Set the key "example" to "foo" */
203     xenstore_write("example", "foo");
204     xenstore_read("example", buffer, 1023);
205     console_write("example = ");
206     console_write(buffer);
207     console_write("\n\r");
208     /* Get info about the console */
209     int length = xenstore_ls("console",buffer,1023);
210     console_write("console contains:\r\n  ");
211     char * out = buffer;
212     while(length > 0)
213     {
214         char value[16];
215         value[15] = '\0';
216         int len = console_write(out);
217         console_write("\n\r  ");
218         length -= len + 1;
219         out += len + 1;
220     }
221 }
```

Now that the basic XenStore driver is finished, you should run it and see if it works. You start it in the same way as the other test kernels, and tell `xm` to grab the console as soon as it starts:

```
# xm create -c domain_config
Using config file "./domain_config".
Started domain Simplest_Kernel
Hello world!
Xen magic string: xen-3.0-x86_32p

VM name: Simplest_Kernel
example = foo
console contains:
  ring-ref
  port
  limit
  tty
```

The string after the VM name is specified in the `domain_config` file. To check that this is working, try changing this. Because this isn't the simplest possible kernel anymore (it has console and XenStore drivers, even if it doesn't do much with them), let's rename it "SimpleKernel." This is done by editing the configuration file and changing the name line to:

```
name = "SimpleKernel"
```

If you run the kernel again, you should get a different output:

```
# xm create -c domain_config
Using config file "./domain_config".
Started domain SimpleKernel
Hello world!
Xen magic string: xen-3.0-x86_32p

VM name: SimpleKernel
...
```

This shows one of the key features of the XenStore; it can be accessed easily by different components in different domains. The name is written by userspace tools in Domain 0 and then read by our simple kernel.

8.5 Other Operations

The kernel example in the previous section read and wrote different keys to the Domain 0 userspace example. This is because it would not have had the correct permissions to write into the root of the XenStore. You can check the permissions on a particular key using XS_GET_PERMS and write them using XS_SET_PERMS (assuming you have enough privilege to do so).

Permissions on a node are identified by a single character: *'r'* and *'w'* represent read-only and write-only permissions. A node for which you have both permissions is indicated by a value of *'b'* whereas *'n'* indicates no permissions. After this character is the domain ID to which the permission refers. A node that domain 5 has read and write permissions for will be indicated by the string "b5".

The most useful feature of the XenStore that has not been discussed here is the ability to set up watches. Sending a message of type XS_WATCH creates a watch on a key. The payload for this message is a pair containing the path to the key, and the key to monitor. This is like a read or list operation, but does not return immediately. Instead, a response is placed in the queue whenever the watched key is next updated until a corresponding XS_UNWATCH request is sent. This is used by the XenBus mechanism, described in more detail in the next chapter, so

that the front end of a split device driver can wait for the back end to enter the correct state for proceeding with the connection, and vice versa.

The watch system is not, strictly speaking, required. The same effect could be achieved by polling the key periodically. This is fine for small numbers of keys, but does not scale well.

This page intentionally left blank

Chapter 9

Supporting the Core Devices

There are two devices in Xen that can be regarded as "core"—the block and network devices. The block device allows a guest to have a persistent store, preventing the state from being lost between reboots, whereas the network device allows it to communicate with the rest of the world. Between the two, they allow a Xen guest to provide a service to the user of the system.

9.1 The Virtual Block Device Driver

All but the most trivial guest will need to support the block device driver. This driver is used to present an interface to an abstract block device, typically a virtual hard disk. This can be backed by various things: real disks, individual partitions, or even files on a host filesystem.

The boot process of a typical operating system involves initializing the block device driver relatively early on, and then passes control over to some initialization code in userspace. Support for block devices is typically the absolute minimum in terms of hardware support for an operating system, after support for the console.

The Xen virtual block device is a simple abstraction of a general block device. As with most other devices, it uses the I/O ring mechanism atop a shared memory page. The virtual block device supports three operations. The first two are obvious: reading and writing a block (or series of block). The third one is a write barrier, which not all back ends support. The write barrier forces all outstanding writes to be completed, and can be used to implement userspace calls such as `fsync`.

The virtual block device is quite similar to many real block devices (SCSI and SATA device, in particular) in that it supports command-reordering. This means that commands issued to it may not complete in the order in which they are

issued. Although this is useful in a real machine, it is essential for achieving good throughput in a virtualized environment. A number of guests may be accessing the same device at the same time, and reordering their commands may well give a significant speed boost. This is particularly true when a number of guests have read-only access to the same backing store, but it is often applicable in other cases too.

The block device makes heavy use of the grant tables. Each transfer is going to be a multiple of the block size, typically at least several KB. As has been shown with real devices, transferring this amount of data is generally much more efficient if done using DMA transfers than host-based copying. To facilitate this, the domU guest makes the destination page available for access by back end of the driver, which can then use it directly for DMA transfer.

The read and write operations are similar in structure to the lio_listio system call. It takes a list of grant reference and block ranges, and transfers the requested data between the specified memory and device. Each transfer has an ID written by the caller, which is preserved in the response, allowing the guest kernel to easily reorder the responses to correspond to the correct requests. A typical implementation might keep a small array of control structures representing transfers, and set the ID for each request to the index of the associated buffer.

9.1.1 Setting Up the Block Device

As with other split device drivers, the front end should initialize the shared memory page and offer the grant references to the back end. It also allocates an event channel, and passes this to the back end. The XenBus mechanism is used to determine the connection state between the back and front ends, and the XenStore is used to transmit the setup information between the two halves.

When a virtual block device is assigned to a given domain, the XenStore should be populated with some information about it. The first step for the front end is to read the XenStore and find out any information it needs to know about the device. The domain's `device/vbd/0/backend` key in the XenStore will give the location within the XenStore of the back end for the first virtual block device. This contains a few keys that need to be read by the front end before finishing the connection:

`sector-size` contains the size of a block.

`size` contains the number of sectors in the device.

`info` provides some extra information about the device. This will be a number generated by ORing several flags together. Currently supported flags indicate that the device is a CD-ROM (1), removable (2), and read-only (4).

Both the front and back end entries in the XenStore include a `state` entry. The front end sets its XenBus state, and reads the back end's state, whereas the back end does the converse. The aim when attaching the device is for both states to be set to XenbusStateConnected, indicating that the device is connected at both ends.

The front end should not do anything until the back end's state is XenbusStateInitialised. Prior to this, the back end is still opening the requisite devices and populating the back end. A particularly verbose front end implementation might want to output the back end state to the console while it is waiting, but this should generally not take long enough for a user to actually read it.

The front end needs to do two things in order to make the device ready: allocate the shared memory segment and the event channel. The first of these steps can be broken into the following stages:

1. Allocate a free page.
2. Initialize the ring on this page.
3. Initialize the private data elsewhere.
4. Share the page using the grant tables.
5. Enter the grant reference into the XenStore.

The first stage is highly kernel-specific. Your kernel, presumably, keeps a list of free memory pages somewhere, which can be used to allocate an unused one. Assuming that the new_page() function (or macro) gets a new page from this list, Listing 9.1 shows how the ring is prepared. This uses the ring macros discussed in detail in Chapter 6. These initialize the producer and consumer counters associated with the rings.

Listing 9.1: Preparing the shared ring for the block device

```
1 blkif_sring_t * shared = new_page();
2 blkif_front_ring_t ring private;
3 SHARED_RING_INIT(shared);
4 FRONT_RING_INIT(&private, shared, PAGE_SIZE);
```

In a real implementation, the private ring would obviously not be allocated on the stack, because it will be needed for future uses of this device.

After the page is allocated and correctly initialized, it must be set up for sharing with the back end. Don't forget that the grant table uses the machine frame number, not the virtual address. The virt_to_mfn macro can be used to convert from the address to an MFN. The domain ID of the back end is also needed. This is stored in the `backend-id` key in the front end's XenStore tree. In most current configurations, this is 0.

Listing 9.2 assumes the existence of a get_grant_ref () function, returning the next free grant reference. The GRANT_TABLE variable should be the grant table itself, and the backend_domain variable should have been filled in with the value of `backend-id` from the XenStore. This snippet shows how the shared page should be offered for sharing.

Listing 9.2: Sharing the block device ring

```
1 int ref = get_grant_ref();
2 GRANT_TABLE[ref].frame = virt_to_mfn(shared);
3 GRANT_TABLE[ref].domid = backend_domain;
4 wmb();
5 ref->GTF_permit_access;
```

As always, this is example code and not production code. It is assumed that the new_page() and get_grant_ref () functions always succeed. In the real world, where such depressing concerns as finite memory space exist, this may not be the case. If either of these fails, a real driver should return an appropriate error message to the kernel.

The shared ring is now set up, and ready for the back end to use. The ring−ref entry in the front-end's XenStore tree should be set to the value of ref for this. Don't forget that the XenStore is entirely text-based, so a string representation of this number is required.

The next step is to set up the event channel. This is relatively easy; an unbound channel must be allocated and then passed to the back end to complete the connection.

After the ring and event channel have been offered to the back end, the front end needs to set its XenBus state to XenbusStateInitialised. This tells the back end that it is ready to be connected. All the front end has to do now is wait for the back end to set its state to XenbusStateConnected. It should then read the three keys mentioned earlier to determine the geometry of the device, and finish any in-kernel configuration that is needed before the rest of the kernel can use the device. Finally, it sets its own state to XenbusStateConnected and transfer can begin.

From the perspective of the back end, the connection process is fairly similar. It does everything it needs to access the physical device, and then sets its state to XenbusStateInitialised. The front end then does all of the connection steps described earlier, enters the XenbusStateInitialised state, and waits. Now the back end needs to perform its own part of the connection.

This is symmetrical to the front end's operations. The back end must map the offered grant reference and then bind the proffered event channel. Mapping the grant reference requires some space to be allocated in the guest's virtual address space to hold it; performing the grant table mapping operation updates the page table entry for the specified page to point to the shared page, but space must be

allocated in the kernel's virtual address space to prevent it from being mapped over a real page.

Typically,[1] a guest's memory layout has the kernel text near the bottom, followed by the remainder of the memory that can be used by the guest. Xen is mapped in at the top, with some shared data (specifically the MFN to PFN mapping table) read only and the rest no-access. This is optimization, making context switches into the hypervisor more efficient. When switching to Xen, the translations for the hypervisor's memory are already mapped, and become accessible when entering ring 0 via the hypercall. The bit in the middle is available. The guest kernel needs to allocate a page from this range for use when performing the mapping.

Assuming that we can retrieve a free page with spare_page(), the mapping should look something like Listing 9.3.

Listing 9.3: Mapping the shared ring into the back end driver

```
1 blkif_sring_t * shared = spare_page();
2 struct gnttab_map_grant_ref op;
3 op.host_addr = shared;
4 op.flags = GNTMAP_host_map;
5 op.ref = front_end_ref;
6 op.dom = front_end_dom;
7 HYPERVISOR_grant_table_op(GNTTABOP_map_grant_ref, &op, 1);
```

The front_end_ref and front_end_dom variables should have already been filled in from the information provided in the XenStore. The hypercall returns a grant reference, which must be kept around for the cleanup phase later, when the grant is unmapped.

9.1.2 Data Transfer

Each transfer to or from a block device is initiated by the guest domain inserting a request into the I/O ring. The structure of the request is shown in Listing 9.4. This defines which operation should be started, and where the data should go or come from.

Listing 9.4: Block device request structure [from: xen/include/public/io/blkif.h]

```
74 struct blkif_request_segment {
75     grant_ref_t gref;        /* reference to I/O buffer frame
               */
76     /* @first_sect: first sector in frame to transfer (
           inclusive).   */
```

[1] This layout describes 32-bit guests. 64-bit guests have the hypervisor mapped differently.

```
77      /* @last_sect: last sector in frame to transfer (inclusive)
            .      */
78      uint8_t      first_sect, last_sect;
79  };
80
81  struct blkif_request {
82      uint8_t         operation;     /* BLKIF_OP_???
                                       */
83      uint8_t         nr_segments;  /* number of segments
                                 */
84      blkif_vdev_t    handle;        /* only for read/write
            requests          */
85      uint64_t        id;            /* private guest value, echoed
             in resp  */
86      blkif_sector_t sector_number;/* start sector idx on disk (r
            /w only)  */
87      struct blkif_request_segment seg[
            BLKIF_MAX_SEGMENTS_PER_REQUEST];
88  };
89  typedef struct blkif_request blkif_request_t;
```

The ring itself does not contain the data being loaded or stored, just the command queue. Each element in the seg member specifies a grant table reference. This should be thought of as being roughly equivalent to a DMA operation; you specify a bit of memory, tell the device to read from it or write to it, and tell you when it is finished. Exactly the same thing happens with the block device. The main difference is that the memory must be explicitly made available via the grant table mechanism. If you have written a device driver for a device on the other side of an IOMMU, this might be familiar to you.

The location on disk is specified by a pair of sector numbers, representing the start and end of the contiguous region. Be aware, however, that the region may not be as contiguous as it seems. A real disk typically presents a linear interface to the kernel, hiding the messy details of the multiple tracks and heads. The virtual block device is likely to be using a disk that performs these abstractions as the final backing store. It is possible, however, that the virtual block device is backed by an object in the Domain 0 filesystem, giving another layer of abstraction. This file may not be contiguous on disk. For this reason, you should be careful when implementing things like read-ahead caching—making the reads all a bit longer might well not be as cheap as you expect.

A common idiom when interacting with block devices is to maintain a block cache. Loaded blocks are stored in the cache and if they are modified and then flushed they are written back to the disk. A userspace process might have direct access to the physical pages of some of these buffers using mmap() or an equivalent

call, or access to a copy of a subset of the data in a buffer using read() and write() system calls.

In such a system, the first step in performing a read operation on the block device is to allocate and prepare a buffer to receive the block. The rest of this example will assume a 512-byte block size and 4KB (page sized, page aligned) buffers. Listing 9.5 shows how such a buffer would be prepared.

Listing 9.5: Preparing a buffer for reading

```
1 char * buffer = new_buffer();
2 int ref = get_grant_ref();
3 GRANT_TABLE[ref].frame = virt_to_mfn(buffer);
4 GRANT_TABLE[ref].domid = block_backend_domain;
5 wmb();
6 ref->GTF_permit_access;
```

The buffer must be allocated and offered to the back end via the grant table mechanism. The next step, shown in Listing 9.6, is to tell the back end what data to transfer and where it should put it.

Listing 9.6: Reading a block from a block device

```
1  blkif_request_t * RING_GET_REQUEST(private, private->req_prod
       ++);
2  request->operation = BLKIF_OP_READ;
3  request->handle = block_vdev;
4  request->sector_number = block_index;
5  request->id = read_index;
6  request->nr_segments = 1;
7  request->seg[0].gref = ref;
8  request->seg[0].first_sect = 0;
9  request->seg[0].last_sect = 7;
10 RING_PUSH_REQUESTS_AND_CHECK_NOTIFY(private, shouldNotify);
11 if(shouldNotify)
12 {
13     struct evtchn_send event;
14     event.port = block_port;
15     HYPERVISOR_event_channel_op(EVTCHNOP_send, &event);
16 }
```

The first thing to do is get a new request, using the ring macro. Note the use of the private ring here. You update the local request producer counter first, and update the shared one later. The next step is to set up the request. For this request, we are only issuing a single read. The sector_number shows the start sector for the request. Each segment is then used to provide a portion of

a buffer. Each page is divided into eight 512-byte[2] sectors. The first and last sector fields for the request indicate which of these should be used. Setting these to zero and seven, respectively, means that the whole page will be filled. The back end performs a linear read, starting at sector_number and continuing until it runs out of buffers to put blocks in. With the current maximum number of segments (BLKIF_MAX_SEGMENTS_PER_REQUEST) being 11, this gives a maximum read or write size of 44KB in a single operation. Note, however, that the back end may combine requests before passing them on to the hardware, so a pair of reads where the second starts at the end of the first could be combined into a single contiguous read.

When the request has been set up, it must be pushed to the back end. This involves a write memory barrier (to ensure that the request has been committed to RAM) and then updating the request producer in the shared ring to reflect the private value.

The macro used to push the request also checks whether the back end has other pending requests. If it does, there is no need to notify it, because it keeps reading requests until it has done all of them, and then sleeps pending future events. If it is waiting, you need to signal it, which you do using the event channel.

The communication between the back and front ends is asynchronous, so the front end now needs to wait until the data is ready before it can proceed. It might do this by masking events and polling, or by returning and waiting for the event upcall. When either of these conditions is reached, the driver should execute some code that looks a little bit like the snippet in Listing 9.7.

Listing 9.7: Handling the response from a read

```
1 blkif_response_t * response RING_GET_RESPONSE(private, i);
2 if(response->id == read_index)
3 {
4     /* Handle response */
5 }
```

A real implementation, of course, is likely to have several requests in flight at once. To support this, you will likely keep an array around somewhere with some meta-data associated with each request and use this to handle the correct action. Don't forget to check the response ID, because the back end is permitted to reorder the requests. Assuming all went well, the status field of the response is set to BLKIF_RSP_OKAY.

Performing a write operation is almost exactly the same. The operation should be set to BLKIF_OP_WRITE, but apart from that, the steps are exactly the same. The only difference is that the data should be in the buffer before a write is performed.

[2] A different sector size can be defined in the XenStore, but 512 is currently the only one used. Very modern hard disks use 4KB sectors, so this may change in the future.

9.2 Using Xen Networking

Most Xen guests require some networking functionality. This is implemented by the virtual interface driver, which follows the standard ring model used by the block device driver and others. Unlike the block device driver, the network is unlikely to be needed for boot, so it can typically be implemented after much of the rest of the operating system is working in Xen. It is somewhat more important to have a working network in a Xen guest than other platforms because much of the communication between a guest and the outside world goes via the network. A Xen guest may use NFS for storage and X11 for user interaction and not use any local devices other than the network interface.

9.2.1 The Virtual Network Interface Driver

The basic structure of the network interface is quite simple. It uses two I/O rings, one for outgoing packets and one for incoming. These are declared in `xen/include/public/io/netif.h`. The rings are used to transmit instructions, but not data. Data is sent via other shared memory pages, offered via the grant mechanism. Each transmission request contains a grant reference and an offset within the granted page. This allows transmit and received buffers to be reused, preventing the TLB from needing frequent updates.

A similar arrangement is used for receiving packets. The domain U guest inserts a receive request into the ring indicating where to store a packet, and the Domain 0 component places the contents there. Earlier versions of this used the grant table transfer mechanism to move data between domains; the buffer would be moved between the two communicating domains' address spaces whenever a packet was received. This caused a lot of TLB churn, however, which turned out to be detrimental to performance, and so newer versions use copying. The copy operation is handled by the hypervisor, in much the same way that the old transfer operation worked.

9.2.2 Setting Up the Virtual Interface

Mapping the network interface is performed in almost exactly the same way as mapping the block interface. The most obvious difference is that the virtual network interface uses two rings, one for transmitting and one for receiving. This is because network traffic can arrive without being directly requested.

The transmit ring is used in the same way as the block device ring when writing. Requests are enqueued in it with packets to write to the network. The receive ring is used in a similar manner to the block ring when performing a read. The front end should take care that there are always some buffers available in the

receive ring. Whenever a frame is received from the network, it is put in the first available[3] buffer. If there are no receive buffers, frames will simply be dropped.

The rings are set up in exactly the same way as before; the same `ring.h` macros were used to define them. Where the data structures defined by these macros for the block device were prefixed with blkif, the network interface rings are prefixed with netif_tx and netif_rx for the transmit and receive rings, respectively.

The next difference is the contents of the XenStore subtree related to the front and back end devices. The keys related to setting up the communication channel are almost the same. A single event channel is used, but two rings are needed. The grant references of both rings must be exported using the `rx-ring-ref` and tx−ring−ref keys.

The back end includes a `mac` key, containing the MAC address of the virtual interface. This replaces the sector and device size keys in the block device back end, which would have no meaning in the context of a network.

The protocol for communicating between the front and back ends drivers allows a few bits of processing to be offloaded. A few keys are written in the XenStore to indicate what features are supported. These keys all start with feature−.

The most important feature is checksum offloading, which is enabled by default, and is disabled by writing "1" to the `feature-no-csum-offload` key. The front end should set it if it doesn't want to make use of this feature.

Another useful feature is TCP segmentation offloading. This is offered by the back end writing "1" to the `feature-gso-tcpv4` key in its tree, and the front end doing the same thing. This allows the guest to write large TCP (currently only IPv4) packets to the payload and have the hardware split them up. More importantly, it does the same thing in reverse and allows the hardware to reassemble them on arrival into individual large packets that can be transferred efficiently.

9.2.3 Sending and Receiving

The same ring macros used for communicating with the block device are used for talking to the virtual network interface. The first thing to do when sending a packet is to create it in memory somewhere.

An Ethernet frame has five fields: the source and destination MAC addresses, the frame type, the payload, and a checksum. Unless the guest has explicitly requested not to use checksum offloading, the checksum should be left blank. This is primarily an optimization for interdomain communication. It is assumed that data does not become corrupted in memory, at least with the frequency it does on the wire, and so calculating and checking the value of the checksum is superfluous.

[3]This will not always be true. See the NetChannel2 section later in this chapter.

This can save a fair amount of CPU time in a pair of domains communicating over the virtual network interface.

Listing 9.8 shows the structure used to transmit a frame. Unlike block I/O, network I/O often deals with odd sized data. The block device can treat a page as an array of blocks to be read from or written to. The network driver does not have this luxury. It must be able to read (and write) arbitrary length Ethernet frames. For this reason, the location of the frame is identified by a page, and then an offset and length within this page. The page is passed by grant reference.

Listing 9.8: Network interface transmit request structure [from: xen/include/public/io/netif.h]

```
68 struct netif_tx_request {
69     grant_ref_t gref;       /* Reference to buffer page */
70     uint16_t offset;        /* Offset within buffer page */
71     uint16_t flags;         /* NETTXF_* */
72     uint16_t id;            /* Echoed in response message. */
73     uint16_t size;          /* Packet size in bytes.       */
74 };
75 typedef struct netif_tx_request netif_tx_request_t;
```

Older versions of the Ethernet specification limited the payload size to 1518 bytes. Newer versions support "jumbo frames," which can be much larger. The typical size is 9000 bytes, which allows 8KB of data and headers for higher layer protocols, without reducing the effectiveness of the 32-bit checksum too much. Although the original frames fit easily inside one page, jumbo pages do not.

This is a problem for the network interface layer, because it passes a single page in each request. This can be addressed by splitting a request across multiple requests. The NETTXF_more_data flag indicates that this request is followed by another that contains the next part of the packet, and that the two should be treated as a single frame.

There are two flags associated with checksum information. NETTXF_csum_blank indicates that the checksum is blank, and NETTXF_data_validated indicates that the checksum has been calculated to match the data in the buffer. Both of these should be set for interdomain communication, because a blank checksum is a valid checksum in this case.

The final flag, NETTXF_extra_info, is used to indicate that the next request contains some extra info in the form of the structure shown in Listing 9.9. Currently, this is only used for segmentation offloading.

The type is there for future expansion, and currently must be set to XEN_NETIF_EXTRA_TYPE_GSO, if this is being used. Note that the segmen-

Listing 9.9: Network interface extra information

```
1  struct netif_extra_info {
2      uint8_t type;  /* XEN_NETIF_EXTRA_TYPE_* */
3      uint8_t flags; /* XEN_NETIF_EXTRA_FLAG_* */
4      union {
5          struct {
6              uint16_t size;
7              uint8_t type; /* XEN_NETIF_GSO_TYPE_* */
8              uint8_t pad;
9              uint16_t features; /* XEN_NETIF_GSO_FEAT_* */
10         } gso;
11
12         uint16_t pad[3];
13     } u;
14 };
```

tation offloading currently only works with TCP on IPv4, not IPv6. The flags are currently not used.

When sending an IP packet across a network, it is sometime necessary to split it into smaller components and then reassemble it at the other end. On an Ethernet network, the host typically splits the packet up into smaller ones that will fit within an Ethernet frame. If the userspace process has provided the data in a single block, this can be quite inconvenient, because it means an extra copy to insert the packet headers in each block.

Segmentation offload delegates this to the card. The host passes a large packet to the network card, which then splits it up into smaller frames. This requires the network card to have some understanding of the high-level protocol so that the appropriate header can be written to each frame, so that the receiver can reassemble the packet.

This extra info section is used by the front end to provide some information about how the fragment should handle the splitting. The size field indicates the size of the resulting packets, which is typically the TCP maximum segment size. The type should be set to XEN_NETIF_GSO_TYPE_TCPV4 for TCP on IPv4, which is currently all that is supported. The remaining two fields are reserved for future use.

When a packet has been sent, a response is pushed back into the transmit ring. This is very simple, and only includes two fields: id and status. The first is the same value as was passed in with the request, whereas the second is usually NETIF_RSP_OKAY. If there were multiple request slots used for a single request, the first one will have this result, whereas subsequent ones will be filled in with NETIF_RSP_NULL, indicating that their status is handled elsewhere.

There are two possible failure conditions when sending a packet. NETIF_RSP_DROPPED indicates that the packet was dropped and should be resent. The other is NETIF_RSP_ERROR, indicating a more serious error.

Receiving packets is simpler than sending them. The netif_rx_request structure only has two fields: the obligatory id field used to match request and response pairs, and a grant reference (gref) indicating the buffer used for receiving packets. Note that the current version of the protocol only permits single-page buffers.

The response, shown in Listing 9.10, is a little more complicated. This indicates the request to which it is responding, and the offset within the page that the received packet starts. The length of the packet is usually stored in the status field. If an error occurs, this will contain an error value (one of the ones listed earlier) instead. All of the error values are negative, so the sign of this value should be checked to determine what the contents represents. Note that NETIF_RSP_NULL is never used for a receive response. Frames that don't fit in a single page will have the length of the segment in the current page passed back in this field.

Listing 9.10: Network interface receive response structure [from: xen/include/public/io/netif.h]

```
153 struct netif_rx_response {
154     uint16_t id;
155     uint16_t offset;       /* Offset in page of start of
            received packet  */
156     uint16_t flags;        /* NETRXF_* */
157     int16_t  status;       /* -ve: BLKIF_RSP_* ; +ve: Rx'ed pkt
            size. */
158 };
159 typedef struct netif_rx_response netif_rx_response_t;
```

The flags field is almost the same as the equivalent in the transmit request structure. If the frame has come from another domain on the same machine, NETRXF_csum_blank will be set, telling the domain not to worry that the checksum has not been filled in. If checksum offloading is turned on, NETRXF_data_validated will be set indicating that the checksum matches the data. This means that the front end does not need to bother with the checking itself.

For frames bigger than a single page, the NETRXF_more_data flag will be set, and the next receive request in the queue will be used to contain the remainder.

Note that the virtual network interface is not the only solution for interdomain communication, although it is the easiest to use. The XenSocket mechanism, which is under development, includes a mechanism for using a shared-memory transport for communicating between two userspace processes in cooperating domains. If you just need to move a lot of data between two domains, you might

Copying versus Flipping

For the size of data of an average network packet, it is faster to copy the data than suffer the hit from invalidating the cache. If the `feature-rx-copy` key in the back end's XenStore subtree is set to "1", it supports copying instead of transferring the page. If this is not used, the back end swaps the page offered by the grant reference with the one containing the buffer. Depending on where the packet was DMA'd to by the physical interface, this may require a copy operation in the back end.

If copying is allowed (by the front-end also setting its `feature-rx-copy` to "1", a much cheaper operation is performed. The hypervisor-based copy is used. Because the hypervisor has all of machine memory mapped into its address space, it can copy from one domain to another without incurring any MMU-interaction overhead.

consider providing an extension to the conventional POSIX shared memory functions that allows interdomain mappings to be established by userspace programs. Getting the security right is nontrivial, but the performance advantage might be worth it for several classes of virtual appliance.

9.2.4 NetChannel2

Networking is a much harder problem than block I/O for Xen. Block I/O typically works on large quantities of data. The practical minimum is the device block size, which is often of the order of 512B, and many operating systems use a 4KB machine page as the minimum amount of data to be read from or written to a block device to make a unified caching infrastructure easier to implement. Network packets are typically less than 1500B for Ethernet. They may be fragmented, or even smaller, especially if they have gone via the Internet. Block I/O operations are typically much less rapid than network operations as well. If more throughput is required from a block device, this is more likely to be accomplished by increasing the size of the requests, rather than the number.

More importantly, unlike block I/O, the correct destination for a network packet can only be determined after reading the header. This contains the MAC address of the recipient, which is either a specific guest or a broadcast address. This means that the block device can DMA data directly into a waiting buffer offered by the receiving guest. For network interfaces, the interface can DMA the data into a buffer owned by the back end, but it must then read the header and decide where to send the data. It then has to either copy the data or remap the page, neither of which is particularly cheap.

If the network card could read the destination MAC, and use a different receive

queue for each destination, this could dramatically improve performance for back end drivers. In this case, the back end would have to handle two situations:

- Local VM to local VM traffic, which doesn't need to go via the network interface at all
- Broadcast traffic which goes to all domains

Some smart NICs offer this kind of functionality. More advanced ones, such as the Solarflare series, allow safe, isolated access to guests. A similar approach is taken by Infiniband interfaces, which allow userspace programs to directly access the interface. This relies on either an IOMMU, or the device enforcing the memory boundaries. In a paravirtualized environment, the driver component in Domain 0 is responsible for providing the device with a list of allowed machine page ranges, and the guest is then allowed to instruct the device to DMA into any of these.

This approach can give the best performance, because there is no hypervisor nor Domain 0 interaction required beyond the initial setup. It has a few issues, however. Firstly, it limits the number of virtual network interfaces to the number available on the card. This is not a huge problem, because the dom0 guest can reserve one for more traditional virtual interface operations, and fall back to that after those on the card have been exhausted, probably with a system administrator assigning the physical device channels to those domains that had the heaviest network traffic.

The other obvious problem of this approach is that it requires domU guests to have hardware-specific drivers. One of the nice features of Xen is that it can be used as a hardware abstraction layer; a domU operating system can implement the front-end network interface driver and be able to make use of as wide a range of network hardware as Domain 0 (or a specialized driver domain) supports.

This, in itself, is not hugely important. Guests can still fall back to using the generic implementation, and use the native hardware support if they need and want to. The problem comes when you start migrating domains. If you move a domain from a machine with a smart NIC to one without, the guest's driver needs to be able to move from using the native card to using the Xen front end. This means that a native driver can't be used directly; instead, a modified version that wraps the Xen and native interfaces must be written. It would be highly inefficient for every interface of this nature to implement this switching code itself, so a standard interface for plugging in multiple interfaces would be useful.

Another problem comes from the fact that communicating between local VMs by sending packets over an external bus is highly inefficient. The current virtual interface is fairly good for interdomain communication. For local communication, it ignores checksums, for example, because if data is becoming corrupted in main memory, you typically have much bigger problems than the odd damaged packet.

One variation on this approach is for a physical interface to appear to be multiple PCI devices, and incorporate a layer 2 switch. This has, roughly, the same advantages and limitations as a device that permits userspace access.

The existing infrastructure for Xen networking has a number of limitations. The most obvious is that it assumes network interfaces are fairly dumb devices, and does not make it easy to take advantage of advanced features. Checksum offloading is supported, and this is quite useful because it allows checksum calculation and checking to be ignored for local communication. More advanced features are difficult to take advantage of.

Another limitation is that the receive buffer size is fixed. Some incoming packets are likely to be very small, whereas others can be huge. A better design is to provide a mechanism for enqueuing different sized receive buffers, and allowing the back end to select the correct sized one. This also addresses the problem of fragmentation of large packets. Currently, if a network packet is larger than the receive buffer, the back end must split it between two or more buffers and the front end must reassemble it. With variable-sized buffers, the back end could just grab a large receive buffer and use that. Typical usage leaves a number of receive buffers enqueued by the front end, and the best fit one is used by the back. This also makes it easier to take advantage of smart NICs that performed reassembly of fragmented packets themselves.

The continual mapping and unmapping of granted buffers by the back end results in a fair amount of TLB churn. It is more efficient if buffers can be reused. Typically, a receiving guest needs to perform an additional copy (unless the underlying hardware supports Mondrian Memory Protection) to give the userland process access to the data, so buffers become unused relatively quickly.

The NetChannel2 protocol is designed to replace the existing virtual interface, and incorporate all of these enhancements. The existing protocol will still be supported for some time, and because it is simpler, it might be a better choice for a first implementation, although the newer interface is likely to give better performance.

At the time of writing, the NetChannel2 specification is a long way from being finalized. It is expected to be finished at some point in 2008.

Chapter 10

Other Xen Devices

The last few chapters have described some of the most important types of paravirtualized devices available to Xen guests. There are quite a few others available, however. Every guest is likely to want to implement support for the console and block devices, and most will want to include support for the network device. Some, however, will also want to add support for USB devices, virtual framebuffers, and the *Trusted Platform Module* (*TPM*). Xen provides all of these, assuming the guest in Domain 0 has the requisite drivers and the hardware is physically present.

If you need to support a device that does not have an existing virtual device type, you may want to add your own. The end of this chapter contains a number of suggestions for building a good interface that fits within the existing Xen infrastructure.

10.1 CD Support

CD drives are block devices like any other, but they often need some special handling. Seeking on a CD is a very expensive operation (typically requiring around a second, compared to under ten milliseconds for a hard disk), and so the caching strategy is likely to be different. CDs also have a somewhat different structure to most hard disks; they have different sessions and tracks rather than a partition table, for example.

Although CDs are not exported as completely different device types, they do have a flag set identifying them. This allows a guest with a mounted CD to handle it differently than other kinds of block devices.

10.2 Virtual Frame Buffer

For a lot of guests, there is no need to provide a graphical display. A virtual network appliance might provide a Web-based configuration interface, and only need to use the console output for debugging boot-time failures. Other guests might prefer to provide a GUI using an existing network-transparent protocol, such as RDP or X11. For some, however, a local frame buffer is preferable.

For these cases, recent versions of Xen provide a virtual frame buffer device. This is backed by a VNC server[1] in Domain 0, and has a somewhat VNC-like interface. The guest writes data into the frame buffer and then notifies the front end of the area that has been updated. This maps directly to the VNC protocol, which only retransmits parts of the image that it detects have been changed.

The virtual framebuffer device is quite unusual, in that it began life as being only available to HVM guests, and was later modified to allow paravirtualized guests to access it. The original code came from QEMU, and did not provide any paravirtualized interface. Since then (Xen 3.0.2), the virtual device has evolved rapidly.

The framebuffer device, like most others, uses a ring structure to transfer commands between the front and back ends. As with many others, the rings are used to transmit commands, rather than data.

Listing 10.1 shows the structure used for the page containing the mapping. This is particularly unusual for a Xen device interface, because it doesn't include the rings for requests and responses. These are implicitly assumed to be stored in the same page as the control structure. Their location is calculated by adding an offset to the address of the start of the page that stores the control structure.

Listing 10.1: Virtual framebuffer device shared structure [from: xen/include/public/io/fbif.h]

```
95  struct xenfb_page
96  {
97      uint32_t in_cons, in_prod;
98      uint32_t out_cons, out_prod;
99
100     int32_t width;          /* the width of the framebuffer (in
            pixels) */
101     int32_t height;         /* the height of the framebuffer (
            in pixels) */
102     uint32_t line_length;   /* the length of a row of pixels (
            in bytes) */
103     uint32_t mem_length;    /* the length of the framebuffer (
            in bytes) */
```

[1] Of course, this does not have to be the case. It could be mapped directly to an X11 window by a suitable back-end driver, for example.

```
104     uint8_t depth;          /* the depth of a pixel (in bits)
            */
105
106     /*
107      * Framebuffer page directory
108      *
109      * Each directory page holds PAGE_SIZE / sizeof(*pd)
110      * framebuffer pages, and can thus map up to PAGE_SIZE *
111      * PAGE_SIZE / sizeof(*pd) bytes.  With PAGE_SIZE == 4096
             and
112      * sizeof(unsigned long) == 4, that's 4 Megs.  Two
             directory
113      * pages should be enough for a while.
114      */
115     unsigned long pd[2];
116 };
```

The biggest difference between the framebuffer and other devices is the way in which updates are handled. Most drivers either place the data directly in the ring, or include a grant table reference pointing to the data. The framebuffer contains a fairly large amount of data (3MB for 800×600 in 32-bit color). Copying this much data would be very expensive. Equally importantly, the typical use for such a region does not require old versions of it to be maintained. Because of this, the driver keeps the frame buffer statically mapped by both halves of the driver. The front end writes to it and sends the back end notifications of "dirty" (modified) regions. These can then be redrawn on the screen.

Most of the fields in this structure describe the shape of the buffer. The width, height, and depth define the display mode. Unlike real displays, which are limited by the capabilities of the hardware, there are far fewer limits on the dimensions of a Xen display. The entire framebuffer must be able to fit into 4MB, due to the way in which the memory is mapped (although this would be simple to expand in the future). Beyond that, it can be any resolution the back end can display, typically any rectangle that will fit in memory. One thing to note is that there is no way of specifying a palette, limiting displays to some form of "true color," either 16 or 32 bits per pixel.

The line_length and mem_length are used to describe the size of the memory used to store the framebuffer. The line length is usually the number of bytes per pixel, multiplied by the number of pixels per line. It might be slightly longer in some cases, to allow the start of every line to be aligned on a natural (machine word) boundary. Similarly, the memory length is normally the line length multiplied by the number of lines.

The location of the framebuffer in memory is defined by the pd field. This contains the machine frame numbers of one or two page directory entries pointing to the region containing the framebuffer.

Although two rings are defined, one for input and one for output events, currently only one output event is defined: update. In future, an input event might be used to notify the guest that regions are occluded (if the virtual framebuffer is being displayed in a windowing system), and that it doesn't need to update these regions. For now, the front end is free to ignore any message sent from the back end. It should, however, increment the in_cons field in the shared structure to clear them.

The only kind of message that can be sent by the front end is a display update. This defines a rectangle in the framebuffer that contains pixels that have changed. The format of this message is shown in Listing 10.2. This should be sent to the back end after updating the pixels, to indicate which regions need to be updated before the next frame is displayed. A full-screen update can be requested by setting x and y to zero and width and height to the width and height of the virtual screen.

Listing 10.2: Framebuffer update message [from: xen/include/public/io/fbif.h]

```
44 struct xenfb_update
45 {
46     uint8_t type;     /* XENFB_TYPE_UPDATE */
47     int32_t x;        /* source x */
48     int32_t y;        /* source y */
49     int32_t width;    /* rect width */
50     int32_t height;   /* rect height */
51 };
```

This structure is not used directly. Instead, xenfb_out_event union should be instantiated. This will have other message types added to it as they a need for them is identified and they are defined. Messages less than XENFB_OUT_EVENT_SIZE can be added without breaking binary compatibility, because the union currently includes a **char** array of this size for padding. Larger messages can be added breaking binary compatibility, but maintaining source compatibility.

The framebuffer itself is only half of the puzzle. A guest that only exposes a command-line interface can use the console for both input and output. After a graphical user interface is displayed, it is likely that the console input will be inadequate. Although it is simple to implement, it does not allow access to a pointing device nor the capability to detect when keys are held down. These extra functions can be accessed via the virtual keyboard interface, which is the companion of the virtual framebuffer.

The design of the virtual keyboard interface is very similar to the virtual framebuffer—both map a single page for control messages, with a C structure at the start for control variables and the rings implicitly stored at a known offset

within the page. The virtual keyboard interface structure, shown in Listing 10.3, is much simpler than the framebuffer page. It simply contains producer and consumer pointers for the two rings. Again, two rings are defined, although only a single one is used. Messages should only be sent to the back end when requested, and there is currently no message type defined to request a message from the front end, nor any "out" message type.

Listing 10.3: Keyboard device shared structure [from: xen/include/public/io/kbdif.h]

```
114 struct xenkbd_page
115 {
116     uint32_t in_cons, in_prod;
117     uint32_t out_cons, out_prod;
118 };
```

Three different types of message can be sent from the back end: two relating to mouse motion and one relating to the keyboard. Keyboard events are of the form shown in Listing 10.4. As with all other messages, the first field defines the type of the message, and should be inspected first to determine how to interpret the message.

These messages signal the change of state of a key. When a key is pressed, the pressed field is set to one, and then a second message is sent when the key is released with this field set to zero. To extract key strokes, the guest must time the interval between these events occurring and implement its own thresholding for auto-repeat.

Listing 10.4: Keyboard button message structure [from: xen/include/public/io/kbdif.h]

```
55 struct xenkbd_key
56 {
57     uint8_t type;          /* XENKBD_TYPE_KEY */
58     uint8_t pressed;       /* 1 if pressed; 0 otherwise */
59     uint32_t keycode;      /* KEY_* from linux/input.h */
60 };
```

Mouse events can be delivered in one of two ways. Either absolute or relative positions can be given. Absolute positions, if they can be supported, are generally preferable because they allow a user on the host machine to interact with the guest without it "capturing" the mouse. Mouse movement events can be triggered as the mouse cursor moves over the window containing the VM's display, allowing the remote display to act just as any other window in the desktop.

Relative motion has some benefits, allowing the mouse to be used for things

Alternative to the VFB

Some GUI systems support remote display natively. Recent versions of Windows, for example, support the *Remote Display Protocol* (*RDP*), which can be used to export a display to a remote machine. Similarly, most UNIX-like systems (Apple's OS X being the most notable exception) use X11 to display graphics.

X11 was built around the idea of network transparency, and works quite cleanly over a virtual network interface. If the Domain 0 (or remote display host) is running X11, the Xnest program can be used to run a child X server in a window. This can be used by a virtual machine to display its "screen" inside the Domain 0 windowing system. For other systems, it is often possible to run an X server within another windowing system.

The biggest advantage of using X11 over the framebuffer is that you can take advantage of any acceleration features supported by the host windowing system. Even OpenGL (which was also designed to be network-transparent) can be accelerated if the X server supports accelerated indirect GLX.

X11 runs faster locally than over a network by using the MIT Shared Memory Extension to allow clients (applications) to use a shared memory transport for sending large amounts of data to the server (display). For intradomain displays, an implementation of this wrapping the grant table mechanism would provide a significant speed boost.

other than moving a pointer over the framebuffer (for example, rotating an object in 3D space), and so this option is also available. A client that wants to receive absolute positioning should set the `request-abs-update` field in the device's entry in the XenStore.

The message used to indicate an absolute mouse position is shown in Listing 10.5. The only fields in this containing data are the absolute coordinates of the new mouse position. These are measured in pixels, and so will always be between zero and the width set for the framebuffer device.

Relative motion is sent by an almost identical structure, with abs replaced by rel in the x and y coordinate fields. This motion is measured in pixels on the host system, but the guest may adjust mouse sensitivity by applying a scaling factor without causing any problems. The type for relative position messages is XENKBD_TYPE_MOTION.

When the framebuffer and keyboard drivers are both working well, a guest can provide a GUI to users from very early on in the boot process. Because the framebuffer is stored in the guest's memory, it can be reattached after migration, without userspace processes in the guest having to be aware of the migration.

Listing 10.5: Absolute mouse position message structure [from: xen/include/public/io/kbdif.h]

```
62 struct xenkbd_position
63 {
64     uint8_t type;         /* XENKBD_TYPE_POS */
65     int32_t abs_x;        /* absolute X position (in FB pixels)
           */
66     int32_t abs_y;        /* absolute Y position (in FB pixels)
           */
67 };
```

10.3 The TPM Driver

The *Trusted Platform Module* (*TPM*) is a controversial piece of hardware, which provides a number of security-related features. One of the complaints often levelled at TPM is that it removes control of the computer from the user. In many situations involving a hypervisor, this is an advantage, because no single virtual machine should be allowed to have complete control of the system. A hypervisor can use some of the features of a TPM to help enforce isolation of VMs. The *remote attestation* features could also be used to ensure that a remote hypervisor waiting to receive a migrated guest was not compromised.

Guests running atop Xen can also make use of a TPM. The TPM provides a mechanism for storing encryption keys and running encryption algorithms in a way that means that the running program never has the key in memory. Because the hypervisor (and Domain 0 in many configurations) can always access guest's memory, a virtual machine cannot trust that its memory is always secure storage. A user may want to carry a VM around with him, to use the same environment on both trusted and untrusted machines. The guest could use the TPM to ensure that sensitive data could only be accessed while the VM was running on a trusted machine, but still allow less sensitive data to be accessed from insecure machines. A similar mechanism could be used to ensure that the kernel was not tampered with while running on an insecure machine.

The interface to the TPM is quite low-level. All the virtual interface does is provide an abstract way of moving TPM control packets to and from the physical (or emulated) device. The contents of the packets are defined by the TPM specification, and will not be discussed here as it is assumed that anyone implementing TPM support is already familiar with the TPM command protocol.

The TPM driver is quite unusual, in that it does not implement ring buffers at all. The protocol for communicating with the TPM is a strict request-response interface, with only a single request in-flight at a time.

10.4 Native Hardware

A guest running in Domain 0, or a driver domain, is typically expected to provide drivers for a number of native pieces of hardware. The easiest way to do this is to give the domain direct access to the hardware. On some platforms, this is possible, because the external interfaces can be made virtualization-aware. On legacy x86, where Xen originated, this is not the case. It is not possible to provide access to part of a PCI bus, for example, without providing access to all of it, making this option impractical for driver domains.

10.4.1 PCI Support

Xen provides a paravirtualized PCI bus device as a way of implementing device pass-through to paravirtualized domain U guests. The guest may interact with the PCI virtual device as it would with a real PCI device. Only devices that are explicitly exported from Domain 0 are visible to the guest, although in the absence of an IOMMU the usual security concerns remain.

This device is fairly simple. It allows PCI device registers to be written to and read from in a way that gives Domain 0 a chance to intercept them and perform basic bounds checking. The PCI device is designed for more-or-less synchronous operation; a single operation is written into the shared memory structure, which must complete before the next one starts.

The shared memory page used for the PCI device is shown in Listing 10.6. The flags field is used to indicate the state of the operation. This has the XEN_PCIF_active bit set by the front end when an operation has been stored in the op structure. This bit is then cleared when the operation has been completed. The remaining 31 bits are reserved for future use.

Listing 10.6: Virtual PCI device shared info page [from: xen/include/public/io/pciif.h]

```
67 struct xen_pci_sharedinfo {
68     /* flags - XEN_PCIF_* */
69     uint32_t flags;
70     struct xen_pci_op op;
71 };
```

The event channel associated with the device is used to signal the fact that a request or response is waiting; however, the device is designed to be used in a polling manner. When waiting for a response, event delivery (upcalls) should be masked. The device can then test the flag to see if the response is ready, and while it isn't, it can wait for the event using the poll scheduling operation. Note that

use of this device does not require any grant table operations (after the initial setup), making it relatively cheap, because there are no TLB updates required.

The message format for this device is shown in Listing **??**. This allows a single register to be read or written in a specified PCI device. A single message of this format is stored in the op field of the structure on the shared memory page.

Listing 10.7: Virtual PCI device operation [from: xen/include/public/io/pciif.h], label

```
47 struct xen_pci_op {
48     /* IN: what action to perform: XEN_PCI_OP_* */
49     uint32_t cmd;
50
51     /* OUT: will contain an error number (if any) from errno.h
           */
52     int32_t err;
53
54     /* IN: which device to touch */
55     uint32_t domain; /* PCI Domain/Segment */
56     uint32_t bus;
57     uint32_t devfn;
58
59     /* IN: which configuration registers to touch */
60     int32_t offset;
61     int32_t size;
62
63     /* IN/OUT: Contains the result after a READ or the value to
           WRITE */
64     uint32_t value;
65 };
```

Whether the operation being performed is a read or a write is decided by setting the cmd field to XEN_PCI_OP_conf_read or XEN_PCI_OP_conf_write. This decides whether the value field is an input or output parameter. If a write operation is being performed, the value to be written is placed there. If the operation is a read, this field is used to return the value.

Most of the rest of the fields are used to identify the register to be accessed: the domain, bus, and devfn. These represent the 8-bit bus, 5-bit device, and 3-bit function ID used to uniquely identify a PCI device. This identifies a 256-byte device configuration space. The offset is then used to identify where within this configuration space the value should be read or written, with the size indicating the number of bits to read or write.

Note that the quantities provided here are larger than those in the PCI specification. This serves two purposes. First, it makes checking that they are valid

easier; the back end can check them without needing to extract them from a more dense format first. Also, it allows the same structure to be used to support PCI Extended and PCI Express buses to be exported without changing the ABI.

If the command proceeds correctly, the error value is set to XEN_PCI_ERR_success. For incorrect device settings, there are two possible errors. If the device simply does not exist, XEN_PCI_ERR_dev_not_found is returned. If it does, but this guest is not permitted to use it, the error code XEN_PCI_ERR_access_denied is returned. For valid devices, the operation can still be blocked if the offset is out of the valid range, with the error value XEN_PCI_ERR_invalid_offset being returned. Finally, functions that are not recognized by the back end returns XEN_PCI_ERR_not_implemented.

Enumerating PCI roots is typically done via a firmware call. This is not possible under Xen. Instead, this information must be retrieved from the XenStore. The `root_num` key in the device's tree contains the number of roots, and `root-0` represents the first one. These are stored in the form `domain:bus`. These keys should be enumerated and parsed when the driver for the bus is connected, replacing the code that would probe a physical device.

The design of the device is such that it should be able to be inserted relatively easily into any kernel that has an abstraction layer for interacting with the PCI bus. This allows existing drivers to be used unmodified, although care still needs to be taken with DMAs, because the drivers must be aware of the machine page numbers.

10.4.2 USB Devices

Support for USB devices within guest domains is quite useful in a number of contexts. Unlike PCI devices, USB devices are plugged in and unplugged relatively frequently, and so need a much more dynamic mapping.

Xen 2.x supported pass-through of USB devices, but this was not well supported and was eventually removed. The Linux implementation was never moved from the 2.4 series kernels to the 2.6, and so it is no longer an option with Xen 3.

Two options currently exist for providing USB devices to other domains. The first is to simply assign a USB controller (PCI device) to the guest. This works well, because the guest simply interacts with the controller as it would with any other USB controller. There are two major downsides to this, however.

The first problem is that it's a static mapping. You can't plug a camera in to a USB port and use it from one domain and then plug a mouse or USB mass storage device in and use it in another. In principle, you could suspend one domain and resume another, and use this mechanism to swap access to the devices, but it's something of a hack and may cause problems with the controller, which will need to be reinitialized fully every switch.

The second is that the guest has control over the entire USB host controller. If

you are lucky, your machine may have two such controllers, allowing you to assign half of the ports to one guest and half to the other, but it is quite likely that you will have to assign all of your USB devices to a single guest.

The other option is to use USB-over-IP. This encapsulates USB messages in IP packets. Currently, the protocol is only well supported in the Linux kernel; however, it is possible to use this to export USB devices over an Ethernet (or any other IP-capable network) connection. This was originally designed for shared network resources. For example, a USB scanner connected to one machine could be accessed from any other device on the network (assuming access permissions were set correctly).

Because USB-over-IP works over any network connection, it can run over the Xen virtual interface. This is generally faster than a USB device, for interdomain connections, but it is still not ideal.

Even with the fast network connection, there is a fair amount of overhead imposed encapsulating the USB data inside IP. This could be reduced somewhat by putting it directly in Ethernet frames, but a better solution is to send it directly using the grant table mechanism. There is work underway to add a USB virtual device, which is likely to be based around the Linux USB-over-IP code, but it is not finished at the time of writing.

10.5 Adding a New Device Type

The Xen hypervisor is not aware of virtual devices at all. It understands events and shared memory, but everything else is built by agreement by virtual machines running in Xen. A virtual device driver is simply an agreement between two domains that a set of shared memory pages, grant table entries, and events should be understood as having certain semantics. As such, implementing a new device type is simply a matter of defining the protocol, and implementing it on both ends, although placing the back end somewhere other than Domain 0 can require some extra work.

10.5.1 Advertising the Device

All devices with the exception of the console and the XenStore itself are advertised via the XenStore. When adding a new device category, it is customary to create a new subtree in the XenStore for each domain and advertise the device there. All configuration information should be supplied in this tree, including the grant reference of the shared memory page for ring buffers and event channels for asynchronous notifications that may be needed.

Assuming the device follows the conventional split driver model, it needs to advertise two things: the event channel and the grant reference of the shared

memory page. The new device should have a name used to identify itself in the XenStore, such as `newdev`.[2] In each domain that may run a device front end, a `device/newdev` tree should be created, with a numerical ID for each available device. Within this, there should be a `backend-id` field containing the domain hosting the back end and a `backend` key containing the XenStore path to the back end. It should also contain all other information required to set up the device, typically a `ring-ref` key for the grant reference to the page containing the device ring and `event-channel` for the event channel used for notifications. Other keys may be created as required.

A similar structure should be created in the device hosting the back end's `backend/newdev` tree, with one entry for each domain that may contain a front end and one entry in this for each device exported to that domain. This should contain `frontend` and `frontend-id` keys containing the domain ID and the XenStore path to the front end. It should also contain any configuration information required for the back-end driver.

10.5.2 Setting Up Ring Buffers

There are two generally used alternatives to ring buffers. If your device has a one-to-one relationship between requests and responses, you can use the generic ring macros in `ring.h`. These handle the creation, initialization, and accessing of rings.

To use these, you need to define a type representing a request, and one representing a response. If you have more than one type of either, you should use a union type. The size of a union type is the size of the largest element, so this ensures that the slots in the rings are big enough to contain all of your requests and responses. The macro for creating a ring itself creates a union of the request and response type for exactly this purpose. These macros were discussed in Chapter 6.

The other option is to define your own rings. This is the option used by most drivers, the exceptions being the block and network devices. If communications happen in only one direction (with no acknowledgment), or in both directions with different rates, this is the best bet. At first glance, it appears that this is the case for the network device, but this is not the case because the front end needs to send grant references to the back end that are used to transfer data from the back end to the front. Something like the console is a better example, because the amount of data transferred is quite small.

The next thing to decide is what other information needs to go in the shared memory space. The virtual framebuffer uses its page to set the dimensions and color depth of the buffer. The decision on whether data should be stored here or in the XenStore is somewhat cosmetic. It's easier (and faster) to access data in

[2] Obviously, this is a bad name for a real device, but it is used here for the purpose of discussion.

the shared memory page from within the driver, but data in the XenStore can be more easily read or modified by tools. The other down side of storing static data in the shared memory page is that it reduces the amount of space available for the rings.

After you have decided how much, if any, static data should be stored in the shared memory page, and the kind of rings you need, you have one final decision to make. Are your rings going to just contain control messages, or will they also contain data? This generally depends on how much data you need to move between the front and back ends of the device.

If you are moving a lot of data, you have two options. You should either pass a grant reference (or more than one) with each request (or response), or use a large static buffer. The block and network devices use the former mechanism. A block write request, for example, includes a grant reference to the page containing the data to be written. The virtual framebuffer uses the second option, and statically maps the entire buffer into both domains' address spaces.

In some situations, a hybrid solution might be useful. For example, something like an interdomain pipe could allocate a few pages as a static buffer, and have two types of message. The first would indicate a range in the buffer containing the new data and the second would contain a grant reference to a page containing the data. For large blocks of data, the overhead of using the grant table would be lower. For smaller bits, the overhead of copying would smaller.

After you've defined the structures for the shared page, including rings and messages, and put them in the header file, you can start writing the two halves of the driver.

10.5.3 Difficulties

Be careful when defining device control structures. When you define a structure for a device's shared memory page, you are defining a binary interface. You can add extra fields in the XenStore without breaking backward compatibility, but if you add fields in the shared memory page, all guests will need their drivers to be recompiled. If you think this is likely, it might be worth adding a "padding" field to the structure, with a few bytes that can be turned into more useful fields later. The down side of this approach is that you are then wasting space in the shared memory page (where you are likely to only have 4KB in total). If possible, it is better to add new data to the XenStore, rather than the shared memory page. If you expect to need a lot more space in the future, you could consider defining an interface including multiple shared pages, allowing the rings to grow to more than 4KB.

The same is true of fields in the request types. If you are using a union type to define requests and responses, you might consider adding some padding to the union. Listing 10.8 shows an example from the virtual framebuffer device.

This only defines a single message currently, but has two other elements in the union. The first is the type. This is useful to have because the first byte can always be accessed using this. The other is padding, which means that other message types can be added without changing the ABI. The protocol states that an implementation should ignore messages it does not understand; so as long as new messages only implement optional functionality, they should still work.

Listing 10.8: Using a union to reserve space for future message types

```
1    union xenfb_out_event
2    {
3        uint8_t type;
4        struct xenfb_update update;
5        char pad[XENFB_OUT_EVENT_SIZE];
6    };
```

Another thing to look out for is variable sizes. C types such as **int** only define a minimum size, not an absolute size. Worse, the sizes might vary between guest operating systems; some systems define an **int** as 32-bits, and some as 64 on x86-64. Even **long**, which should be the same length as a machine word, can cause problems because it is possible to run 32-bit guests on a 64-bit hypervisor and Domain 0. If possible, you should always use explicitly sized types. If you are sending values such as pointers that can be either 32 or 64 bits it is better to use a 64-bit quantity, so you do not need special cases for different domains.

Defining the ABI is relatively simple after you have picked a good abstraction, as long as you keep these things in mind. Picking a good abstraction is very important, however. A virtual device has the following requirements:

- Ease of implementation
- Good mapping to operating system features
- Capability to take advantage of available hardware

The block device is currently the best example of an interface that has all of these properties. The interface is fairly simple, and just defines a way of moving blocks of data between the front and back ends. It maps nicely to how operating systems perceive block devices, because most use them via an abstraction layer with very similar properties, and it can take advantage of DMA and command reordering on devices that support it. The framebuffer is fairly good at the first two, but fails badly on the last one because it can't even take advantage of the 2D acceleration features that have been standard in graphics cards for over a decade.

10.5.4 Accessing the Device

The front half of the driver should not perform any grant table related hypercalls. The front end should create the grant references, and then pass the grant references via the XenStore to the back end. Although this is not required, it does have the advantage that it simplifies the creation of front-end drivers for HVM guests, which are currently unable to issue grant table hypercalls.[3]

The front end should use the XenStore to find the configuration details. It should write the grant references into the store to allow the back end to perform the mapping, and then connect up the event channel.

Although high-end workstations and servers have supported hot-plugging of devices for some time, this is not the case yet with consumer-grade hardware. In a virtual machine, however, all hardware should be regarded as hot-pluggable. When the virtual machine is suspended, it needs to disconnect all of the front-end drivers from the back ends. An operating system on a physical machine entering suspend mode can usually expect the same hardware to be present when it resumes. A virtual machine can make a weaker assumption: that equivalent hardware will be available.

A virtual machine might be suspended on one machine and then resumed on another (or live-migrated). When this happens, it needs to reinitialize the device drivers. When this happens, the back ends might be different. At the very least, machine frame numbers are likely to be different, and event channel numbers on the remote domain are also likely to be different. If your device does not correctly detach and reattach, it will break suspension and migration for the kernel using it.

Exactly how the front-end driver is structured depends on your kernel. For most device categories, most kernels have a generic set of interfaces that higher levels of the kernel use. A well-designed device interface will match these closely.

10.5.5 Designing the Back End

The front end of the device is generally simpler to design than the back. The back end of most devices is somewhat more complicated. The front side interface is usually designed to have a very close mapping from abstractions used in common operating systems, and so is simple to implement. Part of the reason for this is that the front half needs to be implemented in more places; every operating system that runs in Domain 0 also runs in domain U, but the converse is not true. Similarly, a Xen install has only one dom0,[4] but many domU guests. This means

[3] This is likely to change soon, because several people want to run back end drivers in HVM guests.

[4] In this subsection, the back end is discussed as if it runs in Domain 0. Although this is often true, it is also possible to run drivers in "driver domains," specialized domUs with access to certain physical devices.

that the front side of the interface needs to be implemented more times than the back end, and so should be simpler.

The back end, hopefully, can tie into existing multiplexing features in the operating system. At a fundamental level, the purpose of an operating system is to multiplex the resources of a computer between different processes. A back-end block device driver can choose to use either high-level abstractions such as files, or lower level abstractions such as partitions (if the operating system exposes them to userspace tools). It could even choose something in the middle, such as logical volumes, which might be only slightly higher level than partitions (as in Linux's Logical Volume Manager) or almost as high-level as files (as in Solaris' ZFS).

The network device has a similar choice to make: It could interface with the Domain 0 networking stack anywhere from the Ethernet layer right up to the top of the TCP stack. At the lower layers, you get more flexibility, whereas at the higher layers, you get more performance. Because most guest operating systems already have code for injecting Ethernet packets into Ethernet adapters, it is easier to plug a virtual network interface in at the bottom of the network stack. This decision is slightly different from that made by the block device, because it affects the design of the front end, as well as the back. The block device has a "read/write blocks" interface at the front end irrespective of whether the blocks are being stored directly in a partition, in a logical volume, or in a file atop another filesystem. For the block device, the lower levels give better performance, but greater system administration overhead (it is much easier for an administrator to create a file than a partition); but the choice is deferred until runtime.

Some of the implementation details of the back end will be decided by the design of the interface. If your new device presents a fairly high-level abstraction, it might need to interact with the higher layers of the Domain 0 interface stack. The virtual TPM device, for example, needs to tie in quite closely with the physical device.

Simple interfaces tend to offer more flexibility, at the cost of some performance. The virtual framebuffer is a good example of this. All that this device needs is some mechanism of displaying pixels on the screen. One implementation of the back end does this using X11, a system originally designed for providing virtual framebuffers for running text terminals or simple graphical applications inside windows on graphical terminals connected to UNIX machines. This back end maps pixels on the guest's framebuffer to pixels in a window. Another uses the VNC protocol for remote display. At a lower level, a back end could use a real framebuffer device and draw the pixels directly on the screen. The cost paid for this flexibility is performance. A guest cannot draw complex 3D scenes in the virtual framebuffer without incurring a high CPU cost. Drawing a polygon in 3D space first involves mapping it to a 2D polygon, and then to a set of pixels. Newer graphics cards can do all of this in hardware, and older ones can still do the second stage. Even very primitive graphic hardware had support for operations

such as "bit blitting," transferring a rectangular region from memory to the frame buffer with a bit mask indicating whether each pixel in the destination should be overwritten. Modern hardware generalizes this to an alpha blending function, which replaces the bit mask with a transparency value.

When you have determined where the driver needs to go in the host operating system's abstraction layers, the next step is determining what translations are required. For the network interface, the MAC address needs altering. Each network card has a unique MAC address. This is used for routing Ethernet frames, in the same way that an IP address is used for routing IP packets. The original Ethernet design was a bus network, where each frame was broadcast over the connection. Each network interface would compare the destination address against its MAC address and pass matching packets up to the computer. On a switched network, the first frame sent to each MAC address is broadcast on all ports, and the one that replies is cached so that all future frames to that address are only sent to this port.

When the virtual interface is run in bridged mode, the guest virtual machine has its own virtual MAC address which can send and receive Ethernet frames directly from the network. Because this virtual MAC address is not the same as the physical hardware's MAC, the card must be run on "promiscuous" mode. In this mode, the card delivers all Ethernet frames received to the (Domain 0) operating system. This allows frames addressed to the guest's virtual MAC to be received. This kind of behavior is not common for Ethernet interfaces (although it is often supported for debugging network problems), and so serves as a good example of how a back-end driver may need to provide some slightly unusual functionality.

If at all possible, it is often good to have the device driver run in userspace. The framebuffer back ends (X11 and VNC) do this, which allows the virtual framebuffer device to be ported to a new Domain 0 guest much more easily. Of course, the new guest must support the same userspace APIs as Linux (where the device originated) for this to work, but this is generally more common than a new system supporting Linux kernel interfaces.

This page intentionally left blank

Part III

Xen Internals

This page intentionally left blank

Chapter 11

The Xen API

The Xen API is a somewhat confusing term. Most of this book discusses how to interact with Xen, and so it seems odd to have a single chapter dedicated to the "API." This is because Xen provides two interfaces. One is used by guests, and the other is used by tools. The first is known as the *hypercall API*, and is the focus of most of this book. The other is known as the *Xen API*, or sometimes the *Xen Management API*, and is the focus of this chapter. This chapter will discuss the design of the Xen API, how it is used, and how components of the Xen system use it to communicate.

The Xen API is built atop XML-RPC, although C and Python bindings are available, which are generally easier for developers to use. The full API specification is well over 100 pages, and so a complete discussion is well beyond the scope of this chapter.

The Xen API is used by the userspace components of Xen, such as the `xm` command-line tool to control the system. The `xend` dæmon listens for XML-RPC connections[1] and then performs a number of administrative functions.

The Xen API exports everything that you can do with `xm`. This includes most of the control for a VM's lifecycle. This chapter will discuss the API itself, and how the various layers used to implement it are connected.

This chapter will begin by looking at XML-RPC, the underlying protocol from which the Xen API is developed. It will then discuss how the API interfaces with the rest of the system, including the userspace tools and the `xend` dæmon.

[1] Earlier versions of the dæmon used a custom protocol based on S-expressions. This has been deprecated in favor of the XML-RPC protocol described in this section, which should be stable in all releases after 3.1.

11.1 XML-RPC

If you are already familiar with XML-RPC, you can skip this section. If not, this should serve as an overview of the protocol, rather than a complete reference. It should, however, provide enough detail to gain an understanding of the XML-RPC usage within the context of the Xen API.

11.1.1 XML-RPC Data Types

As with most programming languages, XML-RPC defines a small number of primitive data types, and then allows them to be joined together to produce compound data types. Those of relevance to the Xen API are int double, boolean, dateTime .iso8601, and string. These are used to represent **int**, **float**, bool, DateTime and string types in the abstract API.

Each primitive type is represented by a string contained in a pair of XML tags. A floating point value, for example, might be represented in the following way:

```
<double>3.14159</double>
```

The data must be escaped as valid XML character data. In practice, this limitation only applies to the string type, because none of the others permit any characters that are not valid XML. Within the context of the Xen API, all integers are assumed to be 64 bit.

From these simple data types, more complex ones can be constructed. XML-RPC permits two methods of doing this: structs and arrays. These closely mirror the C compound data types of the same name. Arrays and structs are both quite similar—both contain a list of child elements. Arrays contain an ordered list of children, whereas structs contain an unordered list of key-value pairs.

Unlike C arrays, XML-RPC arrays can have heterogeneous contents. Another key difference from C is that the type of every value is encoded with the value, rather than by the variable containing the value. The following is a valid XML-RPC array:

```
<array>
    <data>
        <value><double>3.14159</double></value>
        <value><int>12</int></value>
        <value><string>Xen is the answer.</string></value>
    </data>
</array>
```

Each array tag must contain exactly one data tag, which may contain any number of values. Structs are similar. Unlike C structs, which have a rigid structure, XML-RPC structs are associative arrays. Each struct has an arbitrary

number of key-value pairs. Keys are character data strings, and values can be any XML-RPC type, including arrays or structs. The following shows a simple struct:

```
<struct>
    <member>
        <name>Answer</name>
        <value><int>42</int></value>
    </member>
    <member>
        <name>Question</name>
        <value><string>To be, or not to be?</string></value>
    </member>
    <member>
        <name>True</name>
        <value><boolean>1</boolean></value>
    </member>
</struct>
```

11.1.2 Remote Procedure Calls

Defining an XML format for structured data is potentially useful, but it's not the core of XML-RPC. RPC stands for remote procedure call, and so there needs to be some mechanism for doing this. The standard is built on top of HTTP; each call and response is an HTTP request and response pair.

The format of the request mirrors the structure of a procedure call. Each call contains the name and arguments of the procedure. The request is sent as an HTTP POST of this form:

```
<?xml version="1.0"?>
<methodCall>
    <methodName>example.function</methodName>
    <params>
        <param>
            <value><string>parameter</string></value>
        </param>
    </params>
</methodCall>
```

The method name is just a string, but convention is to treat them as a dot-separated hierarchy. The response is a similar format:

```
<?xml version="1.0"?>
<methodResponse>
    <params>
        <param>
            <value><string>South Dakota</string></value>
```

```
        </param>
    </params>
</methodResponse>
```

Although the structure is similar, there are more constraints. The response may only contain a single parameter, although this can be a compound data type (array or structure). Because these are all completing over HTTP, they are synchronous. In many cases, this is not ideal. The Xen API defines a second top-level namespace "Async". By prefixing the method name with Async. a Xen API user can access an asynchronous version of the calls.

The asynchronous versions of the calls all return a task ID. This is a unique identifier that can be used later to get the real return value. Two additional calls, Async.Task.GetAllTasks and Async.Task.GetStatus are available to get a list of pending completions and the return value of the tasks, respectively.

11.2 Exploring the Xen Interface Hierarchy

The Xen API is defined as XML-RPC at the lowest level. This replaces the older S-expression interface, providing a language-agnostic interface to the control mechanisms for talking to the Xen dæmon. Although S-expressions are as expressive as XML, and simpler to parse and transform, they lack the benefit of existing libraries. XML-RPC is well-supported by a number of third-party libraries that provide language-agnostic ways of invoking XML-RPC calls.

Figure 11.1 shows how the layers are built up in the control interface. The new `libxen` provides C bindings, allowing management tools to be written that do not require a Python runtime. This is used by `libvirt`, which is an aim to provide a consistent API across different virtualization tools. Currently, `libvirt` supports Xen fairly well, and is adding support for KVM and QEMU as well. Several Linux distributions are using it, rather than working with the Xen API directly, and the GNOME desktop environment now includes a local virtualization manager built on top of `libvirt`.

Earlier versions of `libvirt` spawned instances of the `xm` command and parsed the results. This was problematic, because the output from `xm` was intended to be human-readable, and so it was changed periodically to better present the information. The newer versions use the API directly, and so are likely to be more stable. In some ways, the Xen API makes `libvirt` obsolete, because one of its original goals was to isolate tool writers from having to deal with changes in the output format and commands supported by `xm` and related tools. `Libvirt` still serves the secondary purpose of providing the same interface to different virtualization environments, however.

The Xen API is used to provide a bridge between the low-level dæmons and userspace applications. Most Xen management functions are performed by the

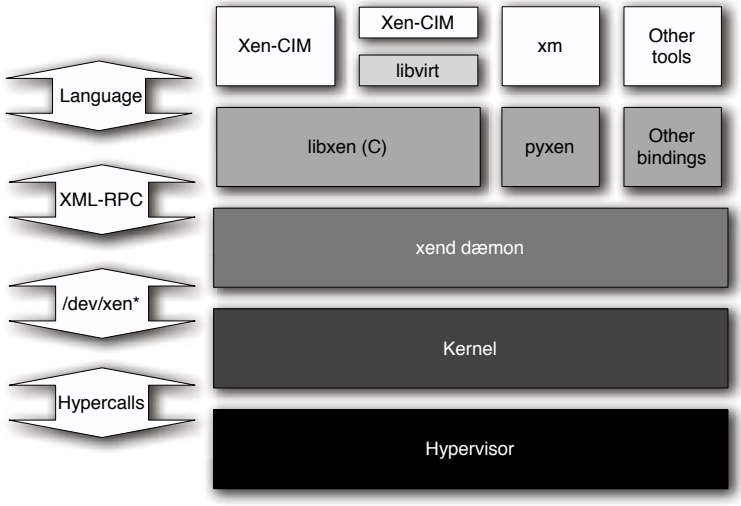

Figure 11.1: The Xen interface hierarchy

`xend` dæmon. This parses the requests from userspace tools and communicates with the Domain 0 kernel to perform management functions.

When a userspace tool issues a command, it is first translated into XML-RPC request by the language bindings, such as `libxen` for C or `pyxen` for Python. This library then sends the XML over a socket to the listening instance of `xend`, which then either handles the request itself or passes it on to the kernel's hypervisor interface and then on to the hypervisor itself.

11.3 The Xen API Classes

The first object in the Xen API that all users have to deal with is the session. Interactions using the API are stateful, and are associated with a particular session. The first task is always to create a new session object. Further interactions are then conducted via this session object.

A session is primarily a means of mapping a user to a host. A session object has a single host associated with it. The host object identifies the physical machine with which the user is communicating. A single management tool might create several different sessions each talking to a different host, allowing a cluster of machines running Xen to be administrated from a central location.

The host has three main types of object associated with it, as shown in Figure

11.2. These correspond to the physical capabilities of the machine, and are used to enumerate the physical block and network devices, and the capabilities of the CPU. Each of these is an attribute with get_* methods associated with it. For example, the host.get_PBDs method returns the set of all physical block devices associated with a given host.

Metrics

In addition to the classes discussed here, most have an associated metrics object. This provides information about the object that would be useful for monitoring tools. For example, every VM has an associated VM_metrics object, which contains the amount of memory, the CPUs, the current state, and a few other bits of information.

The metrics objects are separated from the rest of the system to provide a clean distinction between objects that can be modified by tools and those that simply report on current state. A Xen cluster monitoring tool would be likely to poll all of the metrics objects periodically, and present statistics and warnings to an administrator.

The virtual machines currently running on a given host can be accessed with the host.get_resident_VMs method. This returns a set of VMs running on the system. These then all have the children shown in Figure 11.3, representing the virtual block devices, network interfaces, and TPMs that are associated with the domain and its console.

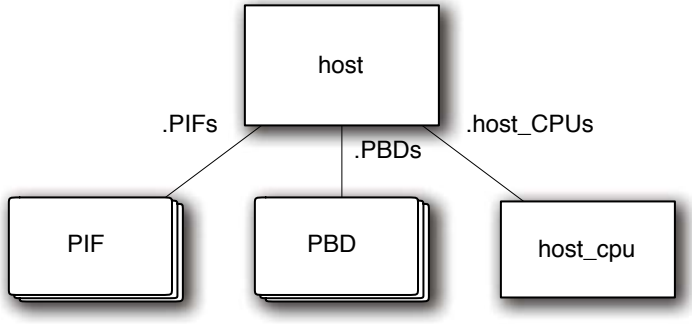

Figure 11.2: Objects associated with a host

Each of these has an associated set of get_ methods, which can be enumerated to determine the configuration information. In principle, the same information

could be extracted directly from the XenStore. This is no longer recommended, since the XenStore layout is not guaranteed to remain stable. It will be likely to remain stable between minor versions, but the Xen API is guaranteed to remain backward compatible from version 1.0. This means that a tool that uses the Xen API to get this data will work with Xen 4.0, but a tool that queries the XenStore directly might not. This becomes more apparent with the NetChannel2 development, which will alter the network interface protocol considerably. If this changes the layout of the XenStore for virtual interfaces then `xend` will perform the translation transparently.

The remaining objects serve as bridges between virtual and physical devices. The various virtual interfaces need to be mapped to physical interfaces, and the virtual block devices to physical block devices. The two objects that are responsible for this mapping are the network and SR (storage repository) objects. These each maintain a list of physical and virtual devices. Every VIF and PIF belongs to exactly one network, and the same mapping is true for block devices to storage repositories.

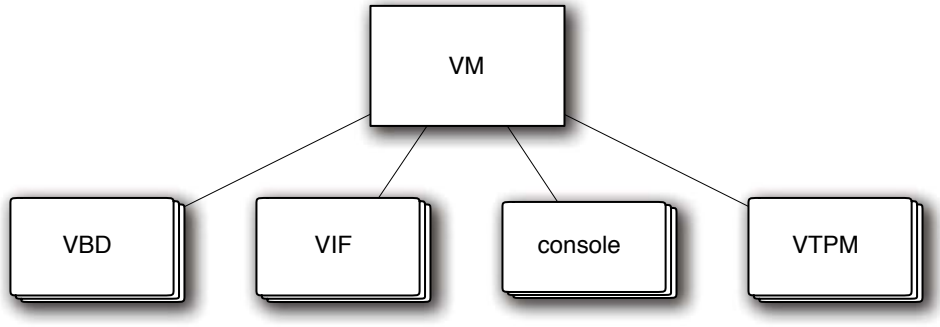

Figure 11.3: Objects associated with a VM instance

11.3.1 The C Bindings

The functions available via the C interface from `libxen` are exposed through a collection of headers with the `xen_` prefix; for example, `xen_vm.h` provides function prototypes and structure definitions for dealing with manipulation of a virtual machine.

The C bindings are designed to closely mirror the underlying interface. They are written by hand to provide the same structures and functions that the XML-RPC interface provides, in a way that can be directly accessed by C programmers.

One thing that makes this difficult is that C does not have a "dictionary" or associative array data type. In most programming languages, these are supported at a fairly primitive level[2] and can be used directly to map the dictionary type used by XML-RPC.

In C, it would be possible to define a dictionary opaque type, with functions for setting and getting key-value pairs. This would not, however, be a programming style that is particularly familiar to C programmers. The usual substitute for a dictionary in C is a **struct**. Unlike a true dictionary, the "keys" in a structure are defined at compile time. Fortunately, the Xen API also defines the keys that may exist in any given dictionary. This allows a **struct** to be constructed for every corresponding dictionary in the API.

In addition to this, a number of enumerated types are defined for symbolic constants sent as strings in the wire protocol. This makes use of the API easier because the standard comparison operator (==) and control structures such as **switch** statements can be used on values from enumerated types, but not on strings. The following snippet of XML represents the reply to a query of the features supported by a given CPU:

```
<array>
    <data>
        <value><string>CX8</string></value>
        <value><string>PSE36</string></value>
        <value><string>FPU</string></value>
    </data>
</array>
```

Listing 11.1 shows the structure used to represent this reply in a prerelease version of the C API. This is a simple array type, which contains a count (because C arrays do not have an accessible size attribute) of the number of elements, and an array of members of the xen_cpu_feature enumerated type. For this reply, the array would be:

```
reply.size = 3;
reply.contents = {XEN_CPU_FEATURE_CX8, XEN_CPU_FEATURE_PSE36,
    XEN_CPU_FEATURE_FPU};
```

This pattern is followed everywhere in the API where a discrete number of possible results are returned. The C bindings are likely to be the ones that require the most work of this nature. Most other languages that would be useful for writing tools either permit direct and easy manipulation of strings, have a "string to atom" built in function, or both. In the final release version of the C bindings, the strings returned by the XML-RPC call are exposed directly. This allows others to be added without modifying the bindings. This pattern is still used where the

[2]Some provide lists of pairs, as a close approximation of a dictionary.

Listing 11.1: CPU feature set structure from `libxen`

```
typedef struct xen_cpu_feature_set
{
    size_t size;
    enum xen_cpu_feature contents[];
} xen_cpu_feature_set;
```

number of returned types is unlikely to change, for example the power state of the virtual machine.

The C bindings do not provide their own way of sending HTTP requests. Users must provide their own functions for actually sending the request. The most common way of doing this is to use `libcurl`, which provides a mechanism for submitting HTTP requests and processing the reply.

Listing 11.2 shows the example call function from the Xen bindings unit testing program. A pointer to this function is passed into `libxen` when initializing the session. Every subsequent call to the API then uses this to retrieve the results. This function performs an HTTP POST operation to the specified URL and delivers the returned data to the specified function, given as an argument.

Listing 11.2: An example function for sending the Xen API call to the server [from: tools/libxen/test/test_bindings.c]

```
78 static int
79 call_func(const void *data, size_t len, void *user_handle,
80           void *result_handle, xen_result_func result_func)
81 {
82     (void)user_handle;
83
84 #ifdef PRINT_XML
85     printf("\n\n--Data to server: ----------\n");
86     printf("%s\n",((char*) data));
87     fflush(stdout);
88 #endif
89
90     CURL *curl = curl_easy_init();
91     if (!curl) {
92         return -1;
93     }
94
95     xen_comms comms = {
96         .func = result_func,
97         .handle = result_handle
98     };
```

```
99
100     curl_easy_setopt(curl, CURLOPT_URL, url);
101     curl_easy_setopt(curl, CURLOPT_NOPROGRESS, 1);
102     curl_easy_setopt(curl, CURLOPT_MUTE, 1);
103     curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, &write_func);
104     curl_easy_setopt(curl, CURLOPT_WRITEDATA, &comms);
105     curl_easy_setopt(curl, CURLOPT_POST, 1);
106     curl_easy_setopt(curl, CURLOPT_POSTFIELDS, data);
107     curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, len);
108
109     CURLcode result = curl_easy_perform(curl);
110
111     curl_easy_cleanup(curl);
112
113     return result;
114 }
```

The C bindings also rely on `libxml`, an MIT-licensed XML-parsing library from the GNOME project. Because it is MIT-licensed, it imposes no restrictions on use nor redistribution in binary form. The C bindings to the API are released under the GNU Lesser General Public License, which imposes no restrictions on the code that links to it, other than that it must be possible for the user to upgrade his version of `libxen`. This is likely to only be a problem for developers creating Xen monitoring appliances, which are intended to be a black box, and do not have a mechanism for installing a modified `libxen`.

11.4 The Function of Xend

The Xen dæmon is responsible for providing the interface between the rest of the userspace tools and the kernel interfaces. Because Xen hypercalls are only permitted from ring 1, it can't issue them directly. Instead, the kernel exports devices with a standardized interface, which are then opened as regular files by the dæmon. It might seem at first glance that `xend` could be implemented entirely in kernel space. It would certainly not be difficult to expose a "perform hypercall" system call, or allow userspace applications to access the Xen device directly. To see why this is not the case, it is necessary to see what `xend` does beyond simply issuing hypercalls.

One of the most obvious responsibilities of `xend` is access control. On the local machine, this is managed by setting the permissions on the socket used to connect to the dæmon as you would any other file, allowing access to be granted to specific users or groups (by default, only root may talk to `xend`). For remote administration, access can be granted to specified SSL client certificates.

Not all management functions require direct interaction with the hypervi-

sor. Some, such as starting various virtual driver back ends, happen entirely in userspace, or with a small amount of hypervisor interaction to establish shared memory regions via the grant tables. If `xend` were entirely kernel-based, it would be very difficult for it to perform many of these functions.

The other major reason for making `xend` a userspace tool is that it makes porting other systems to run as Domain 0 much easier. The kernelspace interface is purposefully kept fairly simple so that as much code can be shared as possible. Although userspace code can trivially be moved between POSIX-compliant operating systems, moving code between kernels is much harder. The Solaris, NetBSD, and Linux Xen ports all run `xend` when being used in Domain 0. The only changes required are relatively minor tweaks to take into account the different filesystem layouts of the guests (such as the correct place in `/var` to store the PID).

Like `xm`, `xend` is written in Python. This imposes an extra constraint on an operating system wanting to run in Domain 0; it must be able to run a Python virtual machine, or include a reimplementation of the management functions. One side effect of a relatively stable API is that the second option is now more readily available. Previously, the interface between `xm` and `xend` was subject to major changes between versions, and was not regarded as public.[3] The introduction of the Xen API means that `xend` could, potentially, be completely replaced without abandoning existing tools. For high-security configurations, a minimal implementation of the API could be written in a language that allowed formal verification, eliminating a potential exploit vector.

`Xend` can be found in the `xen/tools/python/xen/xend` part of the tree. Each of the files in this directory contains an implementation of a part of the Xen API. The Xen API implementation is spread over a small number of files. The `server/XMLRPCServer.py` file contains the code that is first encountered by an incoming request. This sets up the XML-RPC server and maps the XML-RPC commands to various Python methods.

The next step is found in the `XendAPI.py` file. This first performs basic validation of the arguments of XML-RPC methods, such as checking that the referenced objects exist. After this has been done, it acts as a trampoline, and invokes the correct methods for actually handling the request. This trampoline mechanism allows the API and implementation to be changed independently of each other. The same functions are used to implement both the new Xen API, and the old `xm` interface for legacy compatibility.

This has allowed a gradual migration to the Xen API. Individual functions in `xm` were moved over to support the API as it gradually stabilized. With Xen 3.1, the API reached the stage where it supported all of the legacy functions of `xm`, and so could be used as a complete substitute for the old interface. New features added to `xm` will only use the Xen API, and so will be usable from other tools.

[3] Obviously, because both components are open source, it could be inferred relatively easily, but using it was not recommended.

11.5 Xm Command Line

The `xm` ("Xen Master") command is the simplest way of managing Xen. The program itself is written in Python, and can be found in `tools/Python/xen/xm` in the Xen tree. Each command is implemented in a separate source file.

Prior to Xen 3.1, `xm` was the only way of controlling Xen. The program communicated with the back-end dæmon via a private protocol that was not guaranteed to be stable between releases. As of Xen 3.1, `xm` is just another front end to the API, no more important conceptually than any others, although it is still the "standard" way of administrating a Xen system.

The `xm` tool—or, rather, family of tools—is joined together by the code in `main.py`. This contains a number of simple utility functions and common code, as well as the basic command parser. This parser displays usage messages when commands are invoked with no arguments, or when no command is specified. After it has parsed and validated the arguments for a given command, it passes control over to the relevant module.

The `xm` command has always had an interactive mode, invoked by running "`xm shell`" and with the introduction of the Xen API, this was expanded to allow users to talk to the API directly. The following example shows how the metrics for Domain 0 are retrieved (the null UUID always refers to Domain 0 on the local host).

```
# xm shell
The Xen Master. Type "help" for a list of functions.
xm> VM.get_metrics 00000000-0000-0000-0000-000000000000
'bf050307-1ce4-169e-83d7-295703ba3b2a'
xm> VM_metrics.get_record bf050307-1ce4-169e-83d7-295703ba3b2a
{'VCPUs_CPU': {'0': '0'},
 'VCPUs_flags': {'0': ['online', 'running']},
 'VCPUs_number': '1',
 'VCPUs_params': {'cap': '0',
                  'cpumap0': '0,1,2,3,4,5,6,7,8,9,10,11,12,13...
                  'weight': '256'},
 'VCPUs_utilisation': {'0': 0.0014191978269800933},
 'last_updated': <DateTime '20070529T12:14:42' at b7b0fdac>,
 'memory_actual': '486543360',
 'start_time': <DateTime '19700101T00:00:00' at b7b0fe0c>,
 'state': ['running'],
 'uuid': 'bf050307-1ce4-169e-83d7-295703ba3b2a'}
```

The metrics are retrieved with the first call, which gives a handle to the object containing the metrics. Note that in the object-oriented style of the API, the first argument of every method is the "self" object. After a reference to the metrics

object has been retrieved, the data associated with it can be fetched. This is a structure with a number of fields representing data about the running machine.

This shell is a fairly easy way of experimenting with the Xen API. Although at the time of writing, the help command does not provide any information on the Xen API, TAB-completion does work. This allows the methods on objects to be easily listed, or filtered by prefix:

```
xm> host.get_API_version_<TAB><TAB>
host.get_API_version_major
host.get_API_version_minor
host.get_API_version_vendor
host.get_API_version_vendor_implementation
xm> VM<TAB>
Display all 107 possibilities? (y or n)
```

If you are experiencing problems with API calls from a script or application written with one of the bindings, attempting the calls in the `xm` shell can be a quick way of checking that the API actually works as you think it does.

11.6 Xen CIM Providers

One of the driving forces behind the development of a stable API for Xen was the capability to plug in different management interfaces. The *Common Information Model* (*CIM*) is one such desired interface. CIM refers to a family of standards used to define interfaces to management tools. These are not limited to virtualization, and are widely used for a number of management tasks, such as managing complex storage systems. The *Distributed Management Task Force* (*DMTF*) has created a working group to define a standard for integrating virtualization into a CIM stack. The preliminary results of this, the *System Virtualization, Partitioning, and Clustering Working Group* (*SVPC WG*), are being used to construct a CIM provider built on top of the Xen API.

In a sense, the CIM provider is just another language binding. Unlike other bindings, such as `libxen` and `pyxen`, the CIM provider translates between the Xen API and another very high-level, domain-specific language. Although you would use C to write a tool that used `libxen`, you would write something that interfaced with the CIM provider in a general purpose language of your choice. CIM is a data modeling language, not a general purpose programming language.

Why would you want to translate the Xen API into C, and then into CIM, and then into your tool's internal representation, rather than just using the API directly? The main reason is interoperability. Although the Xen API is public, and likely to remain relatively stable, it is not a standard. CIM is a standard,

and is vendor-neutral. A tool that supports CIM is likely to work with other hypervisors, such as the hardware-based hypervisors from Sun and IBM.

CIM versus libvirt

At first glance, CIM and `libvirt` appear to be serving similar purposes. Both provide a hypervisor-agnostic abstraction layer for writing tools. There are a few major differences. CIM is an abstract model, with representations built on XML, and usable in a language-agnostic way. In contrast, `libvirt` is quite closely tied to C.

Another major difference is in the development model. The relevant parts of the CIM standard are being defined by a working group representing multiple vendors. `Libvirt` began life as a wrapper around `xm` and has grown to a more general interface. It is still quite closely tied to the Xen way of doing things, however. It is also designed exclusively for managing virtualization on the local machine, whereas CIM management tools are generally used to organize large numbers of computers and other devices.

Finally, CIM is a large specification, and the parts relevant to virtualization are relatively small. Adding support for managing virtualization to a CIM-aware management tool is relatively simple.

CIM itself is only part of the puzzle. CIM provides an abstract way of describing models of real systems. It is then the job of *CIM-XML* to describe a way of representing the models, and something like *WS-Management*, one of the web services family of standards, to describe a way of interacting with them.

Currently, the Xen CIM provider is being developed out of the main tree.[4] This is likely to continue until the relevant CIM specifications are stable.

11.7 Exercise: Enumerating Running VMs

This section will look at a simple use for the Xen API: enumerating the running virtual machines. First, we will look at how to do this using the C and Python APIs, then examine the XML that is generated, and finally see how `xend` handles the request.

At the highest level, you need to perform the following actions:

1. Establish a connection to `xend` with an associated session.
2. Get the local host for the connection.

[4] Although it can be downloaded from `http://xenbits.xensource.com/ext/os-cmpi-xen.hg`.

3. Get the list of VMs for the host.

To get an idea of how this will work, we'll start by trying it in the `xm` shell:

```
# xm shell
The Xen Master. Type "help" for a list of functions.
xm> session.get_all
['77900a47-c611-8450-28bc-acabab1f8af9']
xm> session.get_this_host 77900a47-c611-8450-28bc-acabab1f8af9
'a86612fa-85b8-ea19-aae7-e60f124d4cb5'
xm> host.get_resident_VMs a86612fa-85b8-ea19-aae7-e60f124d4cb5
['00000000-0000-0000-0000-000000000000']
xm> VM.get_name_
VM.get_name_description   VM.get_name_label
xm> VM.get_name_label 00000000-0000-0000-0000-000000000000
'Domain-0'
```

Here, we cheat slightly getting the session, and get the list of all sessions, because there is only one. Then we get the host from this, and the resident VMs. There is only one VM on the machine being used for this example, Domain 0.

Listing 11.3 shows how you would do the same thing in C. This is a simple program that takes three arguments from the command line: the address of the server and the username and password.

Listing 11.3: Enumerating running VMs in C [from: examples/chapter11/enumerate_vms.c]

```
53 int main(int argc, char **argv)
54 {
55     if (argc != 4)
56     {
57         fprintf(stderr, "Usage:\n\n%s <url> <username> <
               password>\n", argv[0]);
58     }
59
60     url = argv[1];
61
62     /* General setup */
63     xen_init();
64     curl_global_init(CURL_GLOBAL_ALL);
65
66     xen_session *session =
67         xen_session_login_with_password(call_func, NULL, argv
               [2], argv[3]);
68
69     if(session->ok)
```

```
70      {
71          /* Get the host */
72          xen_host host;
73          xen_session_get_this_host(session, &host, session);
74          /* Get the set of VMs */
75          struct xen_vm_set * VMs;
76          xen_host_get_resident_vms(session, &VMs, host);
77          /* Print the names */
78          for(unsigned int i=0 ; i<VMs->size ; i++)
79          {
80              char * name;
81              xen_host_get_name_label(session, &name, host);
82              printf("VM %d: %s\n", i, name);
83          }
84      }
85      else
86      {
87          printf(stderr, "Connection failed\n");
88      }
89      xen_session_logout(session);
90      curl_global_cleanup();
91      xen_fini();
92      return 0;
93  }
```

A couple of helper functions, call_func and write_func, are in the full listing, not shown here. The first is similar to the one shown in Listing 11.2, whereas the second is simply there to reformat the callback from the four-argument version used by `libcurl` to the three argument version required by `libxen`.

Lines 55-60 just ensure that we've got the correct arguments. The address of the server is stored in a global, so that the call_func higher-order function can access it. Lines 62-64 perform initialization of the two libraries that we are using, `libcurl` and `libxen`. Lines 90 and 91 do the corresponding cleanup after we have finished.

In line 66, we attempt to connect to the server with the given username and password. If this works, we proceed. This is the only place where this simple example does any error checking; if we can connect, we will assume that everything else will work. Line 89 is the corresponding destructor for the session object.

In line 73, we retrieve a handle to the host object. Note that the session object is passed in twice. The version of this function from version 0.4 of the API, shipped with Xen 3.0.4, only took two parameters. In this version, the first session is the one we are using for connecting, and the second is the session for which we are looking up the host. The middle parameter is the value in which the host will be returned. This three-argument version allows the host for connections other

than the current one to be looked up. This is not currently useful, because all sessions connected to a single machine will have the same host, but it will be in future versions of the API.

After we have the host, we ask for a set of virtual machines resident on the host (line 76), and then iterate over the resulting set, getting the label (line 81) and then printing it.

Compiling and running the example on the same machine as before gives the following result:

```
# cat Makefile
enumerate_vms: enumerate_vms.c
        @c99 $^ -lxenapi -lcurl -o $@

clean:
        @rm -f enumerate_vms
# make && ./enumerate_vms localhost:8005 user password
VM 0: localhost.localdomain
```

Before we move on, let's take a look at what happens on the wire for some of this. Each of our C function calls corresponds to an XML-RPC request and response. Our initial login corresponds to the XML shown in Listing 11.4. Each of the other functions generates a similar snippet of XML. If you'd like to see exactly what is going on, try modifying the example code to output the XML in the write_func() and call_func() functions.

Listing 11.4: XML generated when logging in

```
OUT:
<?xml version="1.0"?>
<methodCall>
    <methodName>session.login_with_password</methodName>
    <params>
        <param><value><string>user</string></value></param>
        <param><value><string>password</string></value></param>
    </params>
</methodCall>
IN:
<?xml version='1.0'?>
<methodResponse>
    <params>
        <param>
        <value><struct>
            <member>
                <name>Status</name>
                <value><string>Success</string></value>
```

```
19                </member>
20                <member>
21                    <name>Value</name>
22                    <value><string>8a795a7e-1354-f885-0c41-
                          bd4f4ea991fd</string></value>
23                </member>
24            </struct></value>
25            </param>
26        </params>
27 </methodResponse>
```

Rather than try to do the same thing in Python, we will take a look at how `xm` implements the list command. The first thing that needs to be done is to set up the session object, and log in. When using the Xen API, this is done as shown in Listing 11.5.

Listing 11.5: Setting up the session in `xm` [from: tools/python/xen/xm/main.py]

```
2465                server = XenAPI.Session(serverURI)
2466                username, password = parseAuthentication()
2467                server.login_with_password(username, password)
2468                def logout():
2469                    try:
2470                        server.xenapi.session.logout()
2471                    except:
2472                        pass
2473                atexit.register(logout)
```

After this, it calls the xm_list() function. Much of this performs parsing of arguments. It then calls getDomains(), the relevant part of which is shown in Listing 11.6.

Listing 11.6: Getting domain info in `xm` [from: tools/python/xen/xm/main.py]

```
751         dom_recs = server.xenapi.VM.get_all_records()
752         dom_metrics_recs = server.xenapi.VM_metrics.
                get_all_records()
753
754         for dom_ref, dom_rec in dom_recs.items():
755             dom_metrics_rec = dom_metrics_recs[dom_rec['metrics
                    ']]
```

This gets all of the VM records and metrics, and iterates over them. Most of the remainder of this function handles reformatting the data.

11.8 Summary

You have now seen how to interact with the Xen API from C and Python, and how the API works on the wire, making it easy to use from any language with XML-RPC bindings. You have seen how `xend` and `xm` interact within the Xen system, both how they communicate with each other, and where they fit into the rest of the system.

The `pyxen` and `libxen` bindings for Python and C can be used to create monitoring and management tools in Python and C, and other languages can use the API directly, or have bindings written for them relatively easily. Abstract documentation of the interfaces can be found in the `docs/xen-api` part of the tree. At the time of writing, this compiles to a 167-page PDF, which gives developers a lot of information about the functioning of the API. The code for the bindings is in the `tools` subtree, as is `xm`, which provides some examples of how to use the Python interface.

This page intentionally left blank

Chapter 12

Virtual Machine Scheduling

One of the core features of Xen is multitasking. The hypervisor is responsible for ensuring that every running guest receives some CPU time. As with a multitasking operating system, scheduling in Xen is a tradeoff between achieving fairness for running domains and achieving good overall throughput. Some other constraints apply in Xen, due to the nature of some of its uses. One common use for a hypervisor environment is to provide virtual dedicated servers to a variety of customers. These are likely to have some form of service level agreement associated with them, and so it is necessary to ensure that no customer receives less than his allocated amount of CPU, and, for billing purposes, to track when he receives more.

Scheduling for a system like Xen shares some concepts with an operating system that provides an $N : M$ threading library. In such a system, the operating system kernel schedules N threads (typically one per physical context), which a userspace library multiplexes into M userspace threads. In Xen, the kernel threads are analogous to VCPUs and the userspace threads represent processes within the domain. In a Xen system, there can even be another tier, because the guest domain can have userspace threads running on top of it. This gives a potential of up to three schedulers between a thread and the CPU:

1. Userspace threading library mapping userspace threads to kernel threads
2. Guest kernel mapping threads to VCPUs
3. Hypervisor mapping VCPUs to physical CPUs

The hypervisor scheduler, sitting at the bottom of this stack, needs to be predictable. The layers above it will make assumptions on the behavior of the underlying scheduling, and will make highly suboptimal decisions if these are

invalid. This leads to bad, or unpredictable, behavior for processes in the running domains. The design and tuning of the scheduler is one of the most important factors in keeping a Xen system running well.

12.1 Overview of the Scheduler Interface

Xen provides an abstract interface to schedulers. This is defined by a structure that contains pointers to functions used to implement the functionality of the scheduler. Listing 12.1 shows this interface. Readers familiar with static object-oriented languages such as C++ or Java will find this somewhat familiar as a method of defining interfaces.

Listing 12.1: Interface to a Xen scheduler

```
1  struct scheduler {
2    char *name;              /* full name for this scheduler  */
3    char *opt_name;          /* option name for this scheduler*/
4    unsigned int sched_id;   /* ID for this scheduler         */
5
6    void   (*init)           (void);
7    int    (*init_domain)    (struct domain *);
8    void   (*destroy_domain) (struct domain *);
9    int    (*init_vcpu)      (struct vcpu *);
10   void   (*destroy_vcpu)   (struct vcpu *);
11   void   (*sleep)          (struct vcpu *);
12   void   (*wake)           (struct vcpu *);
13   struct task_slice (*do_schedule) (s_time_t);
14   int    (*pick_cpu)       (struct vcpu *);
15   int    (*adjust)         (struct domain *,
16                            struct xen_domctl_scheduler_op *);
17   void   (*dump_settings)  (void);
18   void   (*dump_cpu_state) (int);
19 };
```

When adding a new scheduler, it is necessary to create one of these structures pointing to the newly implemented scheduling functions, and to add it to a static array of available schedulers. At boot time, the correct scheduler can be selected by specifying an argument to the hypervisor. The hypervisor reads "sched={scheduler}" from the list of boot parameters and attempts to match "scheduler" to the opt_name of the schedulers defined as described in Listing 12.1.

Not all of the functions defined by the structure need to be defined for any given scheduler. Any initialized with a NULL pointer are simply ignored. The

simplest valid scheduler would set almost all of the functions to NULL, although this might not actually be very useful.

Current versions of Xen include two schedulers, the older *Simple EDF* (*SEDF*) and the newer *Credit Scheduler*. SEDF is more stable, as it has had longer to undergo testing, but has a few limitations that are causing it to be gradually phased out in favor of the Credit Scheduler.

Adding a new scheduler is something that requires modification to the hypervisor sources and recompilation. This is not as much of a problem as it may appear. Generally, each scheduler can be separated out into its own source file, and the only modification required to the rest of the Xen sources is to add it to the list of available schedulers at the top of `scheduler.c`. This makes it relatively easy to maintain a scheduler outside the main Xen tree.

As well as a list of available schedulers, the `scheduler.c` file contains all of the scheduler-independent code. Each of the functions in the scheduler definition structure has an analog in this source file, which performs any general operations and then calls the scheduler function (if one exists). For example, the schedule() function defined in this file deschedules the running domain and then calls the do_schedule() function for the current scheduler. This returns the new task to be scheduled and the time for which it should run. The generic code then sets a timer to fire at the end of the designated quantum, and runs the new task.

Unlike most of the other scheduler functions, do_schedule() is not optional. All of the other functions are called via a macrom which tests for a non-NULL value and returns 0 if one is found; however, this one does not. This means that a scheduler that does not implement do_schedule() will crash the hypervisor.

Adding a new scheduler is not something that most users of Xen are likely to need to do; however, it is possible that some users of Xen may require scheduling beyond that provided by the existing schedulers. Although the Credit Scheduler is highly configurable, it is not always possible to coerce it to a particular set of needs. If you find this to be the case, you may want to add your own.

Even if you never modify a Xen scheduler, you are likely to find that understanding how it works is beneficial to ensuring that your own guests make effective use of the paravirtualized environment in which they run.

12.2 Historical Schedulers

The two schedulers currently present in Xen are not the oldest ones. Earlier versions of Xen included *Borrowed Virtual Time*BVT, *Atropos* and *Round Robin* schedulers.

The BVT scheduler attempted to give an equal share of runtime to all domains, modified by an administrator-defined weighting. This was the recommended scheduler for Xen 2. It suffered from some problems scheduling I/O

intensive domains, although it was possible to tweak some parameters to avoid certain pathological cases. It also lacked a non-work-conserving mode, making it unsuitable for some uses.

The BVT scheduler works using the concept of virtual time, which elapses only while the domain is scheduled. To allow weighting of VCPUs, BVT increased the virtual time at a rate that is configurable on a per-domain basis. A domain with a low weight would have a smaller virtual time than one with a bigger weight after running for the same amount of wall time. Whenever a scheduling decision needs to be made, the scheduler picks the runnable VCPU that has the earliest effective virtual time. This alone gives a fairly close approximation of a round robin scheduler. The feature that accounts for the word "borrowed" in the name is the capability of domains to "warp." Each domain has the ability to set its virtual time to some point in the past, within a range defined by the administrator.

Each VCPU has an effective and actual virtual time associated with it, with the effective virtual time being calculated by subtracting the warp time from the actual time. There are two limits placed on warping: the maximum amount of time a VCPU can run warped for, and the maximum amount of time it can warp. When a domain requires low latency, it enters the warping state, which makes the scheduler more likely to select it, until it has elapsed its warp time. It must then wait another configurable interval until it is allowed to warp again.

The Atropos scheduler, also from Xen 2, provided soft realtime scheduling. Unlike BVT, which dealt with weights, Atropos guaranteed that each domain would run for n milliseconds every m milliseconds of real time. This was good for latency-sensitive virtual machines, but it was not ideal for CPU throughput. The scheduler did not permit overcommitting of CPU resources. Each CPU was guaranteed a fixed allotment of CPU time, and the remainder was shared out evenly. This made it much harder for a domain to receive CPU "bursts." It is quite common for a virtual (or physical) machine to spend some of its time using almost no CPU, and then some using as much as it has available. This kind of workload did not work well with the Atropos scheduler.

For each VCPU, Atropos maintained a record of the end of its deadline and the amount of time it would be allowed to run before this (that is, the amount of time it had to run every interval, minus the amount of time it had run this interval). Runnable domains were kept in a queue, ordered by deadline. Whenever the scheduler was invoked, it would perform the following steps:

1. Subtract the amount of time for which the domain had just run from its remaining runtime, and if this were zero, move it from the run queue to the waiting queue.

2. Move back to the run queue any domains in the waiting queue that were due to run again.

3. Calculate a new scheduler interrupt time. This could be caused by a domain in either queue. Domains in the waiting queue with very short periods could need to be scheduled before any of the ones in the run queue.

4. Return the domain at the head of the run queue and the time calculated in the previous step.

Realtime scheduling is a difficult problem for Xen, due to the nature of virtualization. It is somewhat difficult to design a realtime scheduler on a single machine without compromising throughput too much. The two (or sometimes three) tier nature of Xen makes it even harder. A solution would likely require some close cooperation between the hypervisor and kernel schedulers, with the kernel scheduler registering wake-up deadlines with the hypervisor. Even this, however, would only allow best-effort realtime scheduling. Soft realtime scheduling is important for a lot of tasks, particularly those on the desktop that involve media recording or playback.

The final historical scheduler was the round robin implementation. Unlike the other two, this was not intended for production use. It was a simple example of the scheduler API that could be used for demonstration purposes. It had a fixed-length quantum that, in the absence of a guest voluntarily yielding its CPU time, each domain would be run for a fixed quantum in order.

Work Conserving

The term *work conserving* is used to describe a scheduler that permits the CPU to run at 100% if any virtual machine (or process, in the case of operating system schedulers) has work to do. A non-work-conserving scheduler imposes a hard limit on the amount of CPU time a given process can consume. Work conserving schedulers are preferred in a lot of cases, because they allow the most efficient usage of the CPU. In some situations, it is desirable to limit the total amount of CPU time a VM can consume—for example, to conserve power or for billing purposes. In these cases, a non-work-conserving scheduler is desirable.

12.2.1 SEDF

The Simple *Earliest Deadline First* (*EDF*) scheduler is the older of the two current Xen schedulers. It is no longer in active development, and is likely to be phased out in the future.

This scheduler works by saying that each domain is set to run for an n ms slice every m ms, where n and m are configurable on a per-domain basis. This scheduler

then picks the VCPU to run which has the closest deadline. For example, consider the following three domains:

1. 20ms slice every 100ms
2. 2ms slice every 10ms
3. 5ms slice every 10ms

Initially, domains 2[1] and 3 have the earliest deadlines for starting their quanta, because they both need to be scheduled within 10ms. Domain 3 has the earliest deadline for starting its quantum, because it must be run in the next 5ms, whereas domain 2 can wait for 8ms.

After domain 3 has run, its next deadline moves into the future, to beyond that of domain 2. These two are scheduled periodically for about 80ms, until domain 1 has to be run. It then wants to take control of the CPU for 20ms. Note that this is longer than the period of the other two domains, which would cause them both to miss their allotments.

At this point, a special case in the code is reached. This detects that allowing domain 1 to run for its maximum slice would mean that other VCPUs would miss theirs. In this case, the SEDF scheduler reduces the allocation so that it terminates in time for the next deadline.

12.2.2 Credit Scheduler

On recent versions of Xen, the *Credit scheduler* is used by default. Each domain has two properties associated with it, a weight and a cap. The weight determines the share of the physical CPU time that the domain gets, whereas the cap represents the maximum. Weights are relative to each other; if all domains have a weight of 128, this has the same effect as giving all domains a weight of 256. In contrast, the cap is an absolute value, representing a proportion of the total CPU that can be used.

By default, the Credit Scheduler is work-conserving. Given two virtual machines with priorities of 128 and 256, the first gets half as much CPU time as the first while both are busy, but can use the whole CPU if the second is idle. The cap is used to force a non-work-conserving mode. If all domains have a cap, and the sum of all caps is below the total CPU capacity, the scheduler does not run any domains for some of the time.

The Credit Scheduler uses a fixed-size 30ms quantum. At the end of each quantum, it selects a new VCPU to run from a list of those that have not already

[1]The SEDF scheduler actually works on VCPUs, rather than domains. For illustrative purposes, we will assume that each domain has a single VCPU.

exceeded their fair allotment. If a physical CPU has no underscheduled VCPUs, it tries to pull some from other physical CPUs.

Whether a CPU is over- or underscheduled depends on how it has spent its credits. Credits are awarded periodically, based on the priority. Consider the following example domains:

1. Priority 64, cap 25%.
2. Priority 64, no cap.
3. Priority 128, no cap.

At the start of a scheduling interval, the first two domains will have 64 credits, whereas the last will have 128.[2] Although all CPUs have work to do, they will be scheduled in a round robin manner. Eventually, the first two domains will be out of credits, and the third one will get all of the CPU to itself for a little while.

If the last domain is idle, the first and second will get equal shares of the CPU until the first has reached its cap of 25%. At this point, the second VCPU continues to run. By doing this, it quickly exhausts its allowance of credits, and is moved into the "overscheduled" queue in the next accounting process. Meanwhile, the other VCPUs continue to accrue credits. At the next accounting, they will be classified as "underscheduled."

Any new allocation of credits that occurs while domain 1 is capped takes this into account, and divide the credits that would be awarded to domain 1 between the other two. This means that the priority of a domain should not be larger than the percentage of the CPU allocated to its cap, or slightly surprising results will occur.

The scheduler ticks every 10ms, subtracting credits from the running VCPU, and caps the minimum number of credits as the number that would be achieved by a process running for one complete time slice having started with no credits.

This minimum value has little effect on the scheduling algorithm. If one VCPU is getting enough runtime to be exceeding the minimum threshold, the others must be either capped or idle. Because idle and capped VCPUs are ignored when determining the allocation of credits, the running VCPU will get more credits than it otherwise would, balancing out the drop. When the other VCPUs have work to do again, they will be factored into the credit allocation and the currently running VCPU will be throttled back to its fair share.

[2]The scheduler actually allocates credits based on a function of the weight, rather than directly as the number given by the weight. These numbers are used for illustration only.

12.3 Using the Scheduler API

Device Scheduling

Access to the CPU is not the only thing that needs to be scheduled. A machine performing a lot of I/O could potentially slow down other domains—for example, by causing a lot of disk seeks, which would reduce total disk throughput.

The hypervisor scheduler is only responsible for controlling access to the CPU. For good (and fair) performance, the back-end drivers need to provide some means of regulating the number of I/O requests that a given domain can perform. Because the hypervisor is not directly aware of the existence of split device drivers, this is delegated to Domain 0 or driver domains.

The scheduler API is defined by the structure from Listing 12.1. This contains four fields that must be filled in, and a number of optional ones. The name field should contain a human-readable name for the new scheduler, whereas the opt_name field contains an abridged version to be used as a selector when specifying a scheduler at hypervisor boot time.

The remaining required field, as mentioned earlier, is the do_schedule() function pointer. This is passed the current time, and returns a struct containing the next VCPU to run, as well as the amount of time for which it should run before being preempted in a task_slice structure, as shown in Listing 12.2.

Listing 12.2: The task slice structure used to indicate the VCPU to run [from: xen/include/xen/sched-if.h]

```
52 struct task_slice {
53     struct vcpu *task;
54     s_time_t     time;
55 };
```

To be able to return a VCPU in this structure, it is necessary for the scheduler to maintain a record of which VCPUs are available at any given time. Whenever a VCPU is started, it is passed to the scheduler's init_vcpu () function. Similarly, when it is destroyed, it is passed to the destroy_vcpu() function. These can be used by a scheduler to keep track of which virtual CPUs are available for scheduling at any given time.

Each vcpu structure has a sched_priv member, which can be used to contain private information for the scheduler, relating to that VCPU. If this is used, then it is the responsibility of the scheduler to destroy it when the VCPU is destroyed.

Some schedulers may treat virtual CPUs different depending on the domain by which they are owned. The domain structure contains a vcpu field, which contains an array of the virtual CPUs owned by the domain. This can be used to ensure fair scheduling between domains, rather than just VCPUs. It should be noted that not all VCPUs for a given domain need to run at the same speed. In extreme cases, it is possible to delegate all scheduling to Xen, and create one VCPU per process in a guest domain. The guest's scheduler is then only responsible for assigning tasks to VCPUs.

The two dump functions are used for debugging purposes. When an administrator requests the current status of the hypervisor, these two functions are called to output the state of the scheduler.

12.3.1 Running a Scheduler

The function pointed to by the do_schedule field of the scheduler structure is called when a scheduler decision is needed. This is called very often, and so should be as short and efficient as possible. The scheduler-independent code that calls this performs the context switch; all that this function needs to do is select the next VCPU to run, and the duration for which it should run. The most common way of doing this is to maintain a run queue and simply take the next available VCPU from the head of this queue.

The do_schedule function should select the next VCPU to run on the current physical CPU. The ID of this CPU can be accessed with the smp_processor_id() macro. This provides the ID of the current physical CPU. It is up to the scheduler itself to store any information it needs to about this physical CPU. Typically, it maintains an array of structures indexed by CPU ID.

After the scheduler is created, the first domain, Domain 0, is assigned to it via the init_domain function. Domain 0 is a special case for scheduling; it has one VCPU for each physical CPU, and these VCPUs are pinned to the corresponding physical CPU, preventing the VCPU being migrated to another physical CPU. The scheduler API does not provide a mechanism for explicitly initializing physical CPUs, and uses this fact as a work-around for needing one.

When a VCPU is added, the init_vcpu function is called. The argument to this function is a vcpu structure, from `xen/include/xen/sched.h`. The processor field of this points to the currently assigned physical CPU. Listing 12.3 shows a snippet from the Credit Scheduler's csched_vcpu_init function, which the Credit Scheduler's init_vcpu field points to. This checks whether the physical CPU associated with the new VCPU is new, and performs per-CPU initialization if it is. There is currently no way of informing a scheduler that a physical CPU is no longer available. Supporting hot-pluggable CPUs requires modifications to the scheduler API.

The scheduler determines how often the do_schedule function is called. When-

Listing 12.3: Checking if a physical CPU needs initializing in the Credit Scheduler [from: xen/common/sched_credit.c]

```
597     /* Allocate per-PCPU info */
598     if ( unlikely(!CSCHED_PCPU(vc->processor)) )
599     {
600         if ( csched_pcpu_init(vc->processor) != 0 )
601             return -1;
602     }
```

ever a domain exhausts its quantum (as defined by the return value of the previous call to this function), or voluntarily yields CPU time, this function is invoked. In earlier versions of the API, there was a periodic timer event associated with each VCPU. Every 10ms of domain virtual time, the VCPU's VIRQ_TIMER virtual interrupt is raised. At the same time as this, the scheduler's tick function was called.

The tick function could be used to trigger periodic accounting functions. The Credit Scheduler used this to invoke the accounting function every tick and the global accounting function every n ticks (where n defaults to 3). This function had just one argument; the index of the physical CPU being ticked. This could be used to tie certain accounting functions to a given CPU, preventing the need for locking. This was used by the Credit Scheduler, which currently always runs the global accounting function on the first CPU. This is a simple way of ensuring that only one version of the function is running at once. Because the tick function was not used by all schedulers, it was removed. Schedulers that require this functionality can schedule their own periodic interrupt.

Most of the other functions within the scheduler are largely to allow internal bookkeeping. The scheduler needs to keep track of:

- Which domains exist
- Which VCPUs are assigned to which domains
- Which VCPUs are sleeping or awake

The scheduler can keep most of this kind of information in the static variables defined in the source file. Some things need to be kept on a per-VCPU basis, however. It is possible to keep a list of structures containing pointers to VCPUs, and scan this whenever scheduler-specific VCPU metadata was required, but it would be incredibly expensive. Instead, the domain and vcpu structures have a sched_priv field. This is a pointer, and can be used to store any information the scheduler chooses to put in there. The Credit Scheduler defines the structure and

macro in Listing 12.4. The structure contains scheduling information about the domain, and is accessed by passing the pointer to the domain into the macro.

Listing 12.4: Per-domain information and accessors from the Credit Scheduler

```
1 struct csched_dom {
2     struct list_head active_vcpu;
3     struct list_head active_sdom_elem;
4     struct domain *dom;
5     uint16_t active_vcpu_count;
6     uint16_t weight;
7     uint16_t cap;
8 };
9 #define CSCHED_DOM(_dom)    ((struct csched_dom *) (_dom)->
    sched_priv)
```

Because the rest of the system is entirely unaware of the existence of this data, it is up to the scheduler to allocate space for it, and free it when the domain is destroyed.

In some cases, a scheduler may need to keep some metadata for each physical CPU. This is slightly more complicated; per-CPU data is stored in a platform-specific way. Each platform defines a set of macros for allocating one instance of a structure for each CPU and accessing these elements. The common scheduler code declares a per-CPU instance of the structure shown in Listing 12.5 called schedule_data.

Listing 12.5: Per-CPU data for scheduling [from: xen/include/xen/sched-if.h]

```
13 struct schedule_data {
14     spinlock_t          schedule_lock;  /* spinlock protecting
           curr        */
15     struct vcpu         *curr;          /* current task
                              */
16     struct vcpu         *idle;          /* idle task for this
           cpu          */
17     void                *sched_priv;
18     struct timer         s_timer;       /* scheduling timer
                            */
19 } __cacheline_aligned;
```

Again, the sched_priv member of this can be used to store scheduler-specific information. Accessing this structure must be done using one of two macros. The per_cpu macro takes two arguments. The first is the name of the structure

(scheduler_data in this case) and the second is the index of the CPU. A common CPU value is the processor element from the VCPU structure, which stores the processor on which the VCPU was last run. The other macro, __get_cpu_var, takes only the name of the structure, and returns the copy that is associated with the current CPU. Semantically, the following two are equivalent:

```
data = per_cpu(scheduler_data, smp_processor_id());
data = __get_cpu_var(scheduler_data);
```

On some platforms, they perform exactly the same operations; however; it is possible that one or the other might have been optimized for a given platform. On 64-bit PowerPC, the Linux version of the latter is more efficient than the former, for example. Because the implementation of these is subject to change between versions and between ports, no assumptions should be made as to their relative efficiency.

12.3.2 Domain 0 Interaction

The HYPERVISOR_domctl hypercall, available from Domain 0, allows access to, and modification of, settings for the scheduler for a given domain. These settings are defined on a per-scheduler basis, as shown in Listing 12.6.

Listing 12.6: Domain 0 scheduler control operation [from: xen/include/public/domctl.h]

```
287 #define XEN_DOMCTL_scheduler_op       16
288 /* Scheduler types. */
289 #define XEN_SCHEDULER_SEDF     4
290 #define XEN_SCHEDULER_CREDIT   5
291 /* Set or get info? */
292 #define XEN_DOMCTL_SCHEDOP_putinfo 0
293 #define XEN_DOMCTL_SCHEDOP_getinfo 1
294 struct xen_domctl_scheduler_op {
295     uint32_t sched_id;  /* XEN_SCHEDULER_* */
296     uint32_t cmd;       /* XEN_DOMCTL_SCHEDOP_* */
297     union {
298         struct xen_domctl_sched_sedf {
299             uint64_aligned_t period;
300             uint64_aligned_t slice;
301             uint64_aligned_t latency;
302             uint32_t extratime;
303             uint32_t weight;
304         } sedf;
305         struct xen_domctl_sched_credit {
306             uint16_t weight;
307             uint16_t cap;
308         } credit;
```

```
309     } u;
310 };
```

Because the settings are passed via a union, adding a new scheduler that responds to this hypercall requires the modification of the `domctl.h` header, which defines it, as well as recompilation of any tools that issue the call. This is not as big a problem as it might first appear. A guest that does not understand the scheduler being used cannot meaningfully interact with it, and so does not need to be aware of the modifications to the hypercall interface required by the scheduler.

The Credit Scheduler has very simple configuration parameters: only a weight and cap can be provided for each domain. SEDF is somewhat more complicated. The difficulty in configuring SEDF was one of the reasons why it was deprecated; it took five settings, all of which could impact performance in related ways. In contrast, the two settings of the Credit Scheduler are largely orthogonal; the weight indicates how much CPU a domain should use when others are competing, and the cap indicates the maximum it can use whether others are competing or not.

At the simplest level, allowing domains to interact with the scheduler allows things like an equivalent of the UNIX `nice` command, adjusting priorities. For some schedulers, it permits more fine-grained control, for example, to give a domain a minimum and maximum CPU allocation.

A scheduler designed specifically for a virtual grid might use a market-based approach, where the cost of CPU time would fluctuate based on demand. Guests could define policies within their domain and buy cycles when they had a lot of work to do, or buy up cheap CPU time to perform periodic bookkeeping work.

Whatever the available options for a given scheduler, they are handled by the function pointed to by the `adjust` field. This takes the scheduler operation from Listing 12.6 and the domain as arguments. How the contents of the structure is interpreted is entirely up to the scheduler.

12.4 Exercise: Adding a New Scheduler

The simplest scheduler to implement, conceptually, is a round robin scheduler without preemption. Because Xen is built around the idea of preemptive multitasking, however, it is actually simpler to implement schedulers that do preemption than those that don't. This example will describe the addition of a trivial scheduler. To simplify matters further, our trivial scheduler will not be SMP-aware; it will not provide any code for dealing with migration of VCPUs or CPU-affinity.

Our code will live in a new scheduler file, `sched_trivial.c`, which must be added to the Xen build configuration. We will not allow any configuration in-

formation, and so there will be no modifications to the hypercall interface nor recompilation of the guest kernel and tools.

Our simple round robin scheduler will not bother to track domains. Each VCPU will be scheduled independently, for a fixed time interval, and then the next one will be run. To do this, it needs to implement three functions: adding and destroying VCPUs and selecting the next one to run.

A list of VCPUs will be maintained as a linked list. New VCPUs will be appended to the list, and whenever a scheduler decision is required the VCPU on the head of the list will be scheduled and moved to the end.

We'll start by creating the run queue, as shown in Listing 12.7. This is a simple singly linked list. Because we are just performing a simple round robin scheduling, we will just use VCPU's scheduler private data pointer to point to the next element.

Listing 12.7: Run queue for the trivial scheduler [from: examples/chapter12/sched_trivial.c]

```
6  /* CPU Run Queue */
7  static struct vcpu * vcpu_list_head = NULL;
8  static struct vcpu * vcpu_list_tail = NULL;
9  unsigned int vcpus = 0;
```

Next, we need to add functions for adding and removing VCPUs from the scheduler's responsibility. This is fairly simple. Our run queue is just a linked list, and we don't store any metadata about the VCPU, so all we need to do is a simple linked list insertion and removal. The functions for doing this are shown in Listing 12.8.

Listing 12.8: Initializing and destroying a VCPU in the trivial scheduler [from: examples/chapter12/sched_trivial.c]

```
12 /* Add a VCPU */
13 int trivial_init_vcpu(struct vcpu * v)
14 {
15     if (vcpu_list_head == NULL)
16     {
17         vcpu_list_head = vcpu_list_tail = v;
18     }
19     else
20     {
21         vcpu_list_tail->sched_priv = vcpu_list_tail = v;
22     }
23     v->sched_priv = NULL;
24     return 0;
25 }
26
27 /* Remove a VCPU */
```

```
28 void trivial_destroy_vcpu(struct vcpu * v)
29 {
30     if(v == vcpu_list_head)
31     {
32         vcpu_list_head = VCPU_NEXT(v);
33     }
34     else
35     {
36         struct vcpu * last = NULL;
37         struct vcpu * current = vcpu_list_head;
38         while(current != v && current != NULL)
39         {
40             last = current;
41             current = VCPU_NEXT(current);
42         }
43         if(current != NULL)
44         {
45             last->sched_priv = VCPU_NEXT(current);
46         }
47     }
48 }
```

After we have the list of VCPUs, the only thing left to do is pick one to run. Listing 12.9 shows how we do this. We always use a 10ms quantum, and then scan along the VCPU list until we find one that is runnable. If we can't find one, we just return the idle task, which is defined by the cross-platform scheduling code on a per-CPU basis.

Listing 12.9: Selecting the VCPU to run in the trivial scheduler [from: examples/chapter12/sched_trivial.c]

```
50 /* Move the front VCPU to the back */
51 static inline void increment_run_queue(void)
52 {
53     vcpu_list_tail->sched_priv = vcpu_list_head;
54     vcpu_list_tail = vcpu_list_head;
55     vcpu_list_head = VCPU_NEXT(vcpu_list_tail);
56     vcpu_list_tail->sched_priv = NULL;
57 }
58
59 /* Pick a VCPU to run */
60 struct task_slice trivial_do_schedule(s_time_t)
61 {
62     struct task_slice ret;
63     /* Fixed-size quantum */
64     ret.time = MILLISECS(10);
65     struct * vcpu head = vcpu_list_head;
```

```
66      do
67      {
68          /* Find a runnable VCPU */
69          increment_run_queue();
70          if(vcpu_runnable(vcpu_list_head))
71          {
72              ret.task = vcpu_list_head;
73          }
74      } while(head != vcpu_list_head);
75      /* Return the idle task if there isn't one */
76      ret.task = ((struct vcpu*)__get_per_cpu(schedule_data)).
            idle);
77      return ret;
78  }
```

This embodies all of the code required to perform scheduling within Xen, in a very simple way. All that remains is to let the rest of the system know that this scheduler is available. This is a two-stage process. The first stage is to create the interface structure, shown in Listing 12.10.

This is an instance of the structure discussed earlier in this chapter, which lets the scheduler-independent code know which functions to call for which purposes.

Listing 12.10: Trivial scheduler definition structure [from: examples/chapter12/sched_trivial.c]

```
80  struct scheduler sched_trivial_def = {
81      .name     = "Trivial Round Robin Scheduler",
82      .opt_name = "trivial",
83      .sched_id = XEN_SCHEDULER_SEDF,
84
85      .init_vcpu      = trivial_init_vcpu,
86      .destroy_vcpu   = trivial_destroy_vcpu,
87
88      .do_schedule    = trivial_do_schedule,
89  };
```

The final step is to edit `xen/common/schedule.c` to contain a pointer to this definition. The modified section is shown in Listing 12.11. Any other schedulers you write should be added in the same way—by adding **extern** declarations of the structures representing the schedulers and then adding a pointer to the structure to the array of schedulers. On boot, the hypervisor kernel iterates over this array, and parses the short name from each structure and sees if the user has selected it in the boot parameters. After this is done, and the source file added to the `Makefile`, it can be built and used.

The example in this section is for illustrative purposes only. It doesn't do SMP-safe operations on the run queue, and so it won't work on multiprocessor

Listing 12.11: Identifying the scheduler to the system

```
55 extern struct scheduler sched_sedf_def;
56 extern struct scheduler sched_credit_def;
57 extern struct scheduler sched_trivial_def;
58 static struct scheduler *schedulers[] = {
59     &sched_sedf_def,
60     &sched_credit_def,
61     &sedf_trivial_def,
62     NULL
63 };
```

systems. On uniprocessor systems, it gives very poor performance, and may fail in unexpected ways. Designing a scheduler is a complex task; it is one of the most-used parts of the hypervisor and you must take into account a number of corner cases. This example is intended to show how a scheduler fits into the Xen hypervisor. The design of a good scheduler is a huge topic, and well beyond the scope of this book.

12.5 Summary

In this chapter, we have looked at some of the historic scheduler implementations in Xen. These all had various flaws that caused them to be deprecated. We then looked at the current implementations, and how they improved matters. Next, we learned how they interfaced with the rest of the system, and examined the separation between the scheduler-independent and scheduler-specific code.

The scheduler-specific code is invoked via a structure that contains function pointers to different parts of the implementation. We saw what each of these functions is supposed to do, and examined some parts of the implementation of the existing schedulers. We saw how the schedulers took commands from Domain 0, or another domain with the correct administration permissions.

Finally, we saw how to add a new scheduler to the hypervisor. The interface to the scheduler is very clean, and so this is relatively easy to do, although designing a good scheduler is still a difficult task.

This page intentionally left blank

Chapter 13

HVM Support

In 2006, both Intel and AMD introduced support for *hardware virtual machines* (*HVM*). Both sets of extensions are conceptually similar, and Xen interacts with both via an abstraction layer. In some ways, the existence of this hardware reduces the need for the approach taken by Xen (and other x86 virtualization solutions), and a new wave of virtualization packages have started to spring up making use of it.

13.1 Running Unmodified Operating Systems

The most obvious benefit of HVM is that it can be used to run Windows as a guest. Supporting Windows as a paravirtualized guest is possible in principle, but since the source code is not publicly available it is not feasible in practice. This ability is particularly important, because one key use for virtualization is to continue to support legacy technology while migrating to a new platform. With HVM, unmodified operating systems can be run in a virtual environment. This support is not limited to Windows; old versions of Linux or other x86 UNIX-like systems, OS/2, or BeOS could also be run as an HVM guest.

Although HVM makes it significantly easier to support virtual machines on x86, the problem of hardware support remains the same. HVM provides a conceptual "ring -1" in which the hypervisor can sit, and allows the trapping of previously untrapable instructions. It does not, however, magically make the hardware virtual. This kind of support can be found in mainframe-segment systems, such as some of IBM's POWER machines, but is not yet available in commodity x86 hardware. Because of this deficiency, a hypervisor for x86 is required to provide emulated devices. Xen implements device emulation by borrowing some code from QEMU.

QEMU and Virtualization

QEMU began life as a CPU emulator. It allowed Linux binaries for one architecture to run on another but still use the native system calls. It was later extended to provide full-system emulation by adding various virtual peripherals, such as a Cirrus Logic GD5446 video card and an NE2000 network card.

QEMU was further extended by the addition of the *QEMU Accelerator* (also known as *KQEMU*). The accelerator turned the emulator into a virtualization system; the "difficult" instructions were still executed in the existing emulation core but unprivileged operations were executed directly on the hardware.

As such, it is possible to use QEMU for virtualization without the need for Xen at all. Using QEMU, however, removes the option of running paravirtual guests and doesn't provide several of the advanced features of Xen such as live migration.

As discussed earlier, emulated devices are a lot slower than paravirtualized ones. Because emulated devices are so slow, they are generally considered a last resort. Xen allows the line between virtualized and paravirtualized to be blurred significantly in HVM mode. An HVM guest still can access the same hypercalls[1] as a paravirtualized guest, so it is possible to write paravirtualized device drivers for otherwise unmodified guests. Because most operating systems provide an interface for writing device drivers, paravirtualized device drivers are often an option even when the kernel source code is not available.

Issuing hypercalls from an HVM guest is performed slightly differently to a fully paravirtualized one. A paravirtualized (PV) guest specifies a location where the hypercall page should be loaded in its ELF header. Specifying the location in the header is obviously not possible in an unmodified HVM guest. Instead, the guest must determine that it is running under Xen at runtime and then retrieve the location of the page. This is done by issuing the **CPUID** instruction. The cpuid instruction returns the vendor ID string of 'XenVMMXenVMM' in **EBX**, **ECX**, and **EDX**, if **EAX** is set to zero. If the value of **EAX** is set to 0x40000000, it can be used to detect the presence of Xen.

When in HVM mode, hypercalls are implemented differently. In PV mode, they are delivered via an interrupt. This is not possible in HVM mode, since interrupts are delivered to the guest kernel. Instead, they must be sent via a **VMEXIT** instruction. The ability to support multiple methods of issuing hypercalls was a major motivation in moving from the old mechanism of issuing the hypercall interrupt directly to the newer method of calling an offset in the hy-

[1] At present, Xen only supports a subset of hypercalls for HVM guests, but support for others is added as a need for them is identified.

percall page. Now, hypercalls can be issued in the same way by guests in PV and HVM environments, although not all of them are yet supported in an HVM environment.

The possibility of true virtualization on x86 leads to the question of whether paravirtualization is still a valid strategy. In principle, it is possible to completely avoid the need to modify the guest. In practice, however, unmodified guests are far from optimal in terms of performance. The biggest advantage of HVM is that it makes porting to Xen an operation that can be completed piecemeal. A fully paravirtualized guest is likely to be faster than an unmodified guest for some time. The biggest performance improvement comes from moving to paravirtualized device drivers. Another important feature to port early on is timekeeping, because a guest that is aware of the distinction between running time and wall time is likely to make more intelligent scheduler decisions. Next, memory management could be ported over to the paravirtualized model, eliminating the need for shadow page tables and giving a further performance boost.

For paravirtualized guests, HVM still provides some advantages. The hypervisor can make use of some of the features provided by HVM implementations in order to accelerate virtual machine execution. For example, an HVM system typically includes a method of saving the entire CPU state in a single instruction. This could be implemented as a small collection of shadow registers or as a fast path to cache, both of which would be significantly faster than manually storing all of the registers for one guest and loading those for another.

Another example is the writable page table assist provided by Xen. It is possible to get much finer-grained control over modifications to the page tables when running on a CPU that supports HVM extensions, allowing these assists to be implemented in a more efficient way.

HVM provides an alternative to paravirtualization, but the best results can be accomplished by using both together.

13.2 Intel VT-x and AMD SVM

As seems to be common in recent years, both AMD and Intel have added incompatible extensions to the x86 ISA in order to provide roughly equivalent functionality: Intel's *Virtualization Technology for x86* (*VT-x*) and AMD's *Secure Virtual Machine* (*SVM*). Both provide a higher privilege mode than ring 0, in which a hypervisor can sit without having to evict the kernel from ring 0. This separation is particularly important on x86-64, because it means that the kernel does not have to run at the same privilege level as the applications, and so no tricks are required to allow it to poke around in their address spaces.

The biggest difference between Intel's VT-x and AMD's SVM comes as an artifact of the way the first chips supporting each are designed. With the Opteron

series, AMD moved the memory controller on-die, whereas Intel kept theirs in a discrete part. Because of the close integration between the CPU and memory controller, AMD was able to add some more advanced modes for handling memory. With VT-x, you simply set a flag that causes page table modifications to be trapped. SVM provides two hardware-assisted modes: shadow page tables and nested page tables.

Shadow page tables have already been described in the context of Xen. The AMD implementation is quite similar. Whenever the guest attempts to update **CR3**, or modify the page tables, the CPU traps into the hypervisor and allows it to emulate the update. This is very similar to the Intel solution, and neither is incredibly fast.

The second mode is known as *Nested Page Tables* (*NPT*), and provides a lot more assistance. NPT adds a higher level to the page table. Each guest is allowed to manipulate **CR3** directly; however, the semantics of this register are modified. The guest sees a completely virtualized address space, and only sets up mappings within the range allocated by a hypervisor. The hypervisor controls the MMU to manipulate the mappings, but does not need to get involved while they are running. This is accomplished by means of a *tagged translation lookaside buffer* (*TTLB*). Each TLB entry has a virtual machine identifier associated with it, and is only valid from within the virtual machine for which it was created. The TTLB means that it is possible to switch virtual machines without needing to flush the TLB, which can potentially give a huge speed improvement.

Newer AMD and Intel chips include a TTLB. In the AMD case, each TLB entry has an *Address Space ID* (*ASID*) associated with it. The ASID is a 6-bit value, giving 64 unique address spaces. Currently, Xen uses two: one for the hypervisor and one for the guests. The TTLB eliminates the need for a full TLB flush on entering and leaving the hypervisor, but still requires one when switching between guests. Work is in progress to use an ASID for each VCPU, allowing up to 63 VCPUs to be run on each CPU without a TLB flush. ASIDs are distributed on a round robin basis. They can be recycled, and this considerably reduces the number of TLB flushes required.

The Intel implementation provides similar results, but in a different way. Addresses are defined by a four-level page table. The first layer is identified by a *Virtual-Processor Identifier* (VPID), similar to an ASID. This is an index into the top layer of the page directory, and can't be changed by the guest. The TLB then simply stores the parts of the whole address that relate to the page and the associated page.

As mentioned earlier, one of the biggest differences between booting in Xen and on a native system is that Xen boots in protected mode, whereas the real hardware starts in real mode. This is somewhat inconvenient for HVM, because it means that unmodified HVM guests are going to expect 16-bit real-mode support.

All x86 chips since the 80386 have included support for a virtual 8086 mode.

This mode was not, however, intended to run real mode operating systems, just real mode applications. As such, a lot of things are not emulated in hardware, and simply trap to the host operating system. Switching to virtual 8086 mode permits the real mode applications to coexist with protected mode ones. Unfortunately, the relatively poor support for real mode instructions means that a lot of things must be emulated when dealing with real mode code running in an HVM guest.

AMD's HVM solution provides a virtual real mode, known as *paged real mode*. This makes it much simpler to support real mode code, because it simply runs natively in the paged real mode environment and causes the same sorts of traps as protected mode code. Intel's, however, does not provice these capabilities. The `tools/firmware/vmxassist` part of the Xen tree contains a partial emulator that handles the traps from virtual 8086 mode.

The VT-x assist code sets up a set of descriptor tables corresponding to the real mode memory layout, then switches to virtual 8086 mode and jumps to the BIOS entry point. The processor then executes 16-bit code, such as the guest boot loader, in virtual 8086 mode. When an invalid instruction (that is, one that is valid in real mode, but not present in virtual 8086 mode) occurs, the processor issues a general protection fault and jumps into the handler set up by the assist code. The handler then emulates the missing instruction.

13.3 HVM Device Support

In the simplest case, HVM guests simply receive a number of emulated devices. In many situations, however, it is beneficial for them to receive pass-through access to real devices. A simple example of this is video, where a single running guest has access to the system's video devices and can run 3D applications.

When virtualizing devices, there are two major things that need to be handled:

1. DMA safety
2. IRQ delivery

AMD's SVM also provides a feature known as the *Device Exclusion Vector* (*DEV*). The DEV is a block of memory used to determine whether a given device is allowed to make DMA transfers into a given block of memory. If a guest is given a piece of hardware, the hypervisor can set up the DEV so that the device cannot be used to touch memory belonging to other domains. Although this makes implementing driver domains safer, it still requires that the guest be aware of virtualization.

To eliminate the requirement for virtualization-aware drivers, an *Input/Output Memory Management Unit* (*IOMMU*) is needed. An IOMMU is similar to the host processor's MMU, but works from the device's perspective. When a device

initiates a DMA operation, it provides a virtual address, rather than a physical one (although it may not be aware of this), and the IOMMU translates the address into a physical address.

Both AMD and Intel have proposed interfaces for IOMMUs for x86 systems. When an IOMMU exists, it is likely that device driver writers are going to want to use it. At the coarsest granularity, an IOMMU can be used to prevent a device controlled by one domain from interfering with another. At a finer granularity, it could be used by the kernel to prevent a driver from writing to memory outside that driver, or even by the driver itself to prevent the device from writing outside the buffers assigned to it.

If the hypervisor is controlling the IOMMU, it will appear to the guest that there isn't one, which means that the fine-grained control is not possible. Work by IBM shows that fine-grained use of an IOMMU can result in up to a 60% performance hit (although this can be improved in some cases), so in a lot of situations it is not unlikely that many users will choose to sacrifice this extra safety. If it is required, the hypervisor needs to provide the guest with access to the IOMMU indirectly. The AMD design is intended to be virtualizable using a shadow page table approach. Updates to the IOMMU's page tables by the guest are trapped and pushed through to the IOMMU when possible. In the paravirtualized setting, some of this overhead can be avoided by integrating IOMMU control with the grant table mechanism.

Intel's *Virtualization Technology for devices* (*VT-d*) goes one step further than simply providing an IOMMU. It also includes a mechanism for interrupt remapping. This works in tandem with VT-x, which makes the system aware of virtual CPUs. By using VT-d in conjunction with VT-x, a device's interrupts are assigned to a virtual, rather than physical, CPU. Each interrupt is uniquely identified not only by its interrupt number, but also by the originator ID (derived from the PCI device ID). The originator ID and interrupt pair is then used to generate a mapping to a virtual CPU number.

Interrupts delivered via the VT-d interrupt remapping mechanism are automatically queued by the hardware, and only delivered when the target VCPU is scheduled. Interrupt remapping is one of the features traditionally provided by Xen. On systems without this capability, the hypervisor must catch all interrupts that are not destined for the currently running guest and translate them into events for later delivery. With VT-d, interrupts can be delivered directly to guests without involving the hypervisor at all.

13.4 Hybrid Virtualization

The new approach being taken in the Linux kernel (for domU) is to adopt a form of hybrid virtualization. The kernel boots in HVM mode, and then checks to see

if it is running under Xen. If it is, it replaces several functions with those related to Xen.

HVM has some advantages from the perspective of a guest. If you execute a **SYSENTER** instruction from ring 3, you get a very fast transition to ring 0. In a fully paravirtualized environment, the hypervisor lives in ring 0, and the kernel in ring 1. This means that the fast system call mechanism does not work. Instead, system calls are executed via an interrupt, which is slightly slower. When running in HVM mode, the kernel runs in ring 0, and the hypervisor in a special mode, so the fast system call mechanism works as expected.

Similarly, in HVM mode, a number of transitions to and from the hypervisor can be avoided. A page fault is a good example of such a transition. Page faults are relatively common on a virtual machine, because memory is typically constrained more than on a real system. A page fault occurs in two situations:

- A guest attempts to access machine memory outside its allocated range.
- A guest attempts to access memory where the "present" bit is not set in the page table entry.

The first case is rare. It should not happen at all; if it happened on a real system, it would be caused by the system accessing memory the BIOS marked as unavailable, or that doesn't exist at all. In this case, something is seriously wrong. The hypervisor may want to give the guest an opportunity to recover, but it is likely that the guest is in an undefined state and so should be killed or debugged.

The second case is much more common, and is typically caused by an application accessing memory that has been swapped out, was lazily allocated,[2] or simply was not allocated to that application. In this situation, the kernel is completely responsible for handling the page fault, by either loading or allocating the memory, or signaling the application. The hypervisor does not need to get involved. In a paravirtualized environment, the hypervisor catches the page fault, and then transitions back to the guest kernel to handle it. In HVM mode, the page fault is delivered directly to the guest. This eliminates two context switches for every page fault, and can make performance significantly faster.

The biggest disadvantage of running a guest in HVM mode is that not all hypercalls are available to HVM guests (although this situation improves with every release). This means that some of the features that make paravirtualized guest operating systems so fast may not be available to a guest running in an HVM domain. For new guests being ported to Xen, incremental porting by starting running as an HVM guest is probably the best approach, although new or experimental kernels may want to start in PV mode and use Xen as a thin hardware abstraction layer to ease development.

[2]Due to the larger cost of page faults, disabling lazy allocation in PV kernels may result in a performance gain.

The first step when porting a guest to run as a Xen-aware HVM guest is to detect whether you are running under Xen.

Listing 13.1 shows how the presence of the hypervisor can be detected from a running guest. Because **CPUID** is an unprivileged instruction, this code can be run from userspace, rather than requiring kernel modification, although it is going to be more useful when integrated into a kernel. Note that when **CPUID** is called with **EAX** set to zero, to retrieve the model ID, the string is returned in the next three registers in the order ebx, edx, ecx, whereas the return string from the call to detect Xen is stored in the order of the registers.

Listing 13.1: Detecting the presence of the Xen hypervisor [from: examples/chapter13/isXen.c]

```
#include <stdio.h>
#include <sys/types.h>
#include <string.h>

typedef union
{
    uint32_t r[3];
    char string[12];
} cpuid_t;

#define CPUID(command, result) \
    __asm __volatile(\
            "CPUID"\
            : "=b" (result.r[0]), "=c" (cpu.r[1]), "=d" (cpu.r
                [2])\
            : "a" (command));

int main(void)
{
    cpuid_t cpu;
    CPUID(0,cpu);
    if(strncmp(cpu.string, "XenVMMXenVMM", 12) == 0)
    {
        printf("Running as a Xen HVM guest\n");
    }
    else
    {
        printf("Running on native hardware or a non-Xen
            hypervisor.\n");
    }
    return 0;
}
```

The **CPUID** instruction, like much of x86, has suffered somewhat from feature creep. Originally, it was used to identify the model of the CPU, and the supported features. Now it is used as a way of accessing some quite detailed information about the CPU, including the layout of the cache and TLB. The instruction writes a set of values to the first four general purpose registers, depending on the value in **EAX** when it is issued.

Accessing the hypercall table from inside an HVM guest is quite simple. First, the **CPUID** instruction is issued with a value of 0x40000002 in **EAX**. This returns the number of pages required for the hypercall area (currently a single page, but this could change in the future) and the number of a *model-specific register* (*MSR*) in **EAX** and **EBX**, respectively.

The kernel then needs to allocate some space to store the hypercall range and map the pages by writing to the MSR (using the WRMSR instruction) the pseudo-physical addresses of the pages to contain the region, in order.

Listing 13.2 shows how Linux performs this mapping, when running as an HVM guest. The kernel allocates pages pages in its own address space, making sure to clear the "no execute" bit on the permissions (the pages are going to be used as jump addresses, so being able to write to them is important). It then finds the pseudo-physical frame numbers for the pages and maps them.

Listing 13.2: Mapping the hypercall page(s) by an HVM guest [from: unmodified_drivers/linux-2.6/platform-pci/platform-pci.c]

```
136     cpuid(0x40000002, &pages, &msr, &ecx, &edx);
137
138     printk(KERN_INFO "Hypercall area is %u pages.\n", pages);
139
140     /* Use __vmalloc() because vmalloc_exec() is not an
            exported symbol. */
141     /* PAGE_KERNEL_EXEC also is not exported, hence we use
            PAGE_KERNEL. */
142     /* hypercall_stubs = vmalloc_exec(pages * PAGE_SIZE); */
143     hypercall_stubs = __vmalloc(pages * PAGE_SIZE,
144                     GFP_KERNEL | __GFP_HIGHMEM,
145                     __pgprot(__PAGE_KERNEL & ~_PAGE_NX));
146     if (hypercall_stubs == NULL)
147         return -ENOMEM;
148
149     for (i = 0; i < pages; i++) {
150         unsigned long pfn;
151         pfn = vmalloc_to_pfn((char *)hypercall_stubs + i*
                PAGE_SIZE);
152         wrmsrl(msr, ((u64)pfn << PAGE_SHIFT) + i);
153     }
```

The hypercall macros all work by multiplying the hypercall number by a constant and using the result as an offset within the hypercall page. If the hypercall "page" grows larger than a single page, the pages must be mapped to a contiguous region in the kernel's address space (which is typically the pseudo-physical address space) in order to work.

This layer of indirection allows the same interface to be used for hypercalls on different platforms. On pure paravirtualized systems, hypercalls are issued using an interrupt, just as system calls were on older x86 systems. Intel VT-x systems use **VMCALL**, whereas AMD SVM systems use **VMMCALL**. The running guest just needs to jump into the hypercall page and the correct mechanism will be used.

After mapping the trampoline page, hypercalls can be issued as if the HVM guest were paravirtualized. The hybrid-virtualized Linux kernel uses a subset of the paravirt ops implementation for Xen. Paravirt ops is an abstract interface from the Linux kernel to various hypervisors, where paravirtualized versions of privileged operations are stored in a structure and invoked based on the detected hypervisor.

13.5 Emulated BIOS

A typical x86 PC contains a BIOS, which provides basic input/output support, and a VGA BIOS, which provides a simple interface to the graphics card. Because these have both been around for quite some time, they are generally expected to exist by an operating system. The BIOS provides basic text-console functionality, so a VGA BIOS is only required for graphical user interfaces. It is not commonly used these days, because it does not provide any hardware acceleration support. Because it is almost always present, however, it does tend to be supported by a fall-back, or emergency display driver, by most operating systems. Even in normal use, it is commonly used to initialize the display, before switching to a device-specific driver.

The BIOS code used by Xen originates with Bochs, an x86 full-system emulator, whereas the VGA BIOS was created to be modular and plug into Bochs and Plex 86, an open source attempt at virtualizing and later paravirtualizing x86.

The PC BIOS is fairly simple in comparison to more modern firmwares, but is still nontrivial to emulate. It must support a fairly wide range of calls. When a guest OS boots, it expects to be able to use the BIOS to communicate with the keyboard, hard and floppy disks, system console, and so on.

The HVM BIOS is a replacement for the front end to the split device driver model. The guest issues a BIOS interrupt requesting data be loaded from the disk, and the "BIOS" then translates the call into a request to the block device. The BIOS interrupt is caught by code taken from QEMU, which emulates the

device's functionality. As with the split driver model, the actual device I/O is handled in the back end (in Domain 0). The only difference is that more effort is required by the "back end" in the HVM case because it must translate from some emulated-hardware specific interface into an abstract for and then back to the concrete form used by the real hardware.

13.6 Device Models and Legacy I/O Emulation

For unmodified guests, there are two possible solutions to device support:

- Provide access to real devices to a specific domain. For example, a sound card could be delegated to a Windows guest without affecting other Xen guests.
- Emulate hardware for which existing drivers exist. This has to be a very fine-grained emulation. The emulator must trap instructions which write to the I/O registers, either using port or memory mapped I/O, and any interrupts which would normally be handled by the BIOS.

Option one is fairly easy, but not very useful a lot of the time. It also generally needs an IOMMU to work securely. An HVM guest sees memory in one way, and a device without an IOMMU sees it in another way. When the guest specifies a DMA transfer, the device reads from or writes to a completely different area of memory to the one the guest expects. The DEV can be used to prevent this from overwriting other domains, but it does not magically make the device behave as the guest expects. An IOMMU can be used to set the same virtual memory mappings for the devices as for the guest, allowing it to work as expected.

The other option, providing emulated devices, is typically easier. There are two ways of doing this: The emulator can be run in Domain 0 and be mapped into the I/O range, or the emulated devices can be run in an HVM stub domain and communicate with Domain 0 via the normal split driver mechanism. The first approach has the advantage that it is easy to implement, but it has a few limitations. The first is that runtime for the emulated devices is accounted to Domain 0, rather than the guest. The second is that it involves running more code in a privileged domain, which gives more potential for security risks.

At the time of writing, Xen is in process of migrating from the first approach to the second. HVM guests have the device emulators running in their own address space, in a memory area reserved by the BIOS, and communicate with the rest of the system as if they were normal split driver model front ends. This has a significant advantage when it comes to porting guests, because it is possible to move to paravirtualized drivers without the need to modify the domain. This can even be done for closed-source operating systems; they can be booted with emulated devices and then move to paravirtualized version after boot. This approach

is quite common on a number of platforms; the OS boots using a lowest-common-denominator set of drivers, then moves to better ones afterward. Several systems, for example, boot using the BIOS or OpenFirmware to access the disk, then switch to direct access drivers supporting higher-speed modes during the boot process.

The "native" device emulators can be found in the `tools/ioemu` part of the tree.

13.7 Paravirtualized I/O

On many full-virtualization systems, the biggest bottleneck is disk I/O. An I/O-intensive application can have the host machine spending 90% or more of its cycles emulating the device, leaving only a small amount for actually doing the work.

The I/O bottleneck is dramatically reduced by replacing the block device driver for the emulated device with a front-end virtual block device driver. This is a relatively simple procedure, after the hypercall page has been mapped. Most of the device setup, however, depends on the XenStore, and so a XenStore driver must be installed first.

The XenStore is usually set up using information in the start info page. On HVM guests, however, there is no start info page, which presents something of a bootstrapping problem. The solution is the HYPERVISOR_hvm_op hypercall, with the HVMOP_get_param command.

Listing 13.3 shows how the event channel and shared page number are determined. From here, the procedure for setting up the XenStore is exactly as it would be for a paravirtualized guest. The same is true of other devices.

Listing 13.3: Setting up the XenStore for HVM guests

```
1     struct xen_hvm_param xhv;
2     xhv.domid = DOMID_SELF;
3     xhv.index = HVM_PARAM_STORE_EVTCHN;
4     if(HYPERVISOR_hvm_op(HVMOP_get_param, &xhv) < 0)
5     {
6         /* Handle error */
7     }
8     xen_store_evtchn = xhv.value;
9     xhv.index = HVM_PARAM_STORE_PFN;
10    if(HYPERVISOR_hvm_op(HVMOP_get_param, &xhv) < 0)
11    {
12        /* Handle error */
13    }
14    xen_store_mfn = xhv.value;
```

One other difference exists, typically below the PV driver layer. An HVM guest does not register a callback for event delivery. Instead, there is a virtual

PCI device provided that is used to communicate with the hypervisor. This device raises an interrupt whenever an event is ready for delivery. The interrupt that should be raised is defined by the guest.

Listing 13.4 shows how the platform PCI driver for Linux sets up this IRQ. After the hypercall has been issued, events are delivered by raising the interrupt, and can be de-multiplexed as before.

Listing 13.4: Setting up the event-delivery IRQ [from: unmodified_drivers/linux-2.6/platform-pci/platform-pci.h]

```
27 unsigned long alloc_xen_mmio(unsigned long len);
28 void platform_pci_resume(void);
29
30 extern struct pci_dev *xen_platform_pdev;
31
32 #endif /* _XEN_PLATFORM_PCI_H */
```

Masking events is done via the shared memory page. Again, this is accessed slightly differently from an HVM guest. A PV guest has the shared info page in its psuedo-physical address space at the start and simply needs to update its page tables to include it. On HVM guests, the shared info page needs to be added to the guest's pseudo-physical address space. Listing 13.5 shows how Linux maps the shared page.

Listing 13.5: Mapping the shared info page for HVM guests [from: unmodified_drivers/linux-2.6/platform-pci/platform-pci.c]

```
81     xatp.domid = DOMID_SELF;
82     xatp.idx = 0;
83     xatp.space = XENMAPSPACE_shared_info;
84     xatp.gpfn = shared_info_frame;
85     if (HYPERVISOR_memory_op(XENMEM_add_to_physmap, &xatp))
86         BUG();
```

With the shared info page mapped, the XenStore driver working, and the event handler configured, paravirtualized device drivers can be used on HVM guests. For Linux, exactly the same source code is used for virtual device drivers in both pure PV and PV-on-HVM kernels.

13.8 HVM Support in Xen

Support for different forms of hardware virtualization within Xen is done via an abstract interface. The hvm_function_table structure contains a number of functions for performing a number of hardware-assisted functions. A single instance of this structure, hvm_funcs, is filled in by the hypervisor for the available features during the boot process.

The first step in initializing this structure is to determine the kind of physical CPU being used. The code in `xen/arch/x86/cpu/common.c` does this initialization for x86 chips. This code interrogates the running CPU and determines the type, then calls the per-vendor initialization routines.

Currently, the only vendors supporting HVM extensions to the x86 instruction set are Intel and AMD. Xen includes CPU-specific code for a number of other manufacturers, including Cyrix (now VIA), Rise (now SiS), and Transmeta, but these models do not provide any virtualization-specific features.

The last line in the init_amd() and init_intel () functions, from the `amd.c` and `intel.c` files in `xen/arch/x86/cpu` respectively, call start_svm() and start_vmx(), respectively. These functions are defined in files under the `xen/arch/x86/hvm` part of the tree, with a subdirectory for each form of HVM assistance.

These two functions have the same general structure. First, they detect the presence of HVM support, and return early if it is not supported. If it is, they fill in the hvm_funcs structure with CPU-specific functions, set the hvm_enabled flag, and return. This flag is used later to determine if HVM is supported on the current system.

One of the most obvious ways in which the two HVM implementations differ is the way in which hypercalls are dispatched. Listings 13.6 and 13.7 show the function used to set up the hypercall page for AMD and Intel HVM domains, respectively.

Listing 13.6: AMD SVM hypercall page setup [from: xen/arch/x86/hvm/svm/svm.c]

```
738     memset(hypercall_page, 0, PAGE_SIZE);
739
740     for ( i = 0; i < (PAGE_SIZE / 32); i++ )
741     {
742         p = (char *)(hypercall_page + (i * 32));
743         *(u8  *)(p + 0) = 0xb8; /* mov imm32, %eax */
744         *(u32 *)(p + 1) = i;
745         *(u8  *)(p + 5) = 0x0f; /* vmmcall */
746         *(u8  *)(p + 6) = 0x01;
747         *(u8  *)(p + 7) = 0xd9;
748         *(u8  *)(p + 8) = 0xc3; /* ret */
```

The first thing to notice about these two listings is how similar they are. They both iterate over the page writing a short sequence of instructions for each entry.

Listing 13.7: Intel VT-x hypercall page setup [from: xen/arch/x86/hvm/vmx/vmx.c]

```
981     memset(hypercall_page, 0, PAGE_SIZE);
982
983     for ( i = 0; i < (PAGE_SIZE / 32); i++ )
984     {
985         p = (char *)(hypercall_page + (i * 32));
986         *(u8  *)(p + 0) = 0xb8; /* mov imm32, %eax */
987         *(u32 *)(p + 1) = i;
988         *(u8  *)(p + 5) = 0x0f; /* vmcall */
989         *(u8  *)(p + 6) = 0x01;
990         *(u8  *)(p + 7) = 0xc1;
991         *(u8  *)(p + 8) = 0xc3; /* ret */
992     }
```

The first instruction moves the hypercall number to **EAX**. The 0xb8 value is the opcode for the **MOV** instruction variant with the **EAX** register as the destination and a 32-bit immediate value as the source. The next word is the hypercall value. Skipping forward to the end of each entry, we get the 0xc3 value, indicating a **RET** instruction, which returns to the call used to enter the page.

Between the **MOV** and **RET**, we get some CPU-specific code. On AMD CPUs, the vmmcall instruction is used, whereas Intel CPUs use the vmcall instruction. For reference, Listing 13.8 shows the equivalent code used when building a 32-bit paravirtualized domain. Here, the transition is accomplished using the **INT** $0x82 instruction, which jumps to the hypervisor's interrupt 82h handler.

Listing 13.8: Hypercall page for 32-bit paravirtualized guest [from: xen/arch/x86/x86_32/traps.c]

```
500     {
501         p = (char *)(hypercall_page + (i * 32));
502         *(u8  *)(p+ 0) = 0xb8;     /* mov  $<i>,%eax */
503         *(u32 *)(p+ 1) = i;
504         *(u16 *)(p+ 5) = 0x82cd;   /* int  $0x82 */
505         *(u8  *)(p+ 7) = 0xc3;     /* ret */
506     }
```

Note that all of the hypercall mechanisms, so far, have been eight or nine bytes. This means that almost three quarters of the space on the hypercall page is "wasted." Even the x86-64 implementation, which makes use of the syscall instruction, is only 14 bytes long—less than half of the 32 bytes allowed. This does not present much of a problem. Even with 32 bytes per entry, this allows

Xen up to 128 hypercalls before a second page is required. The extra space allows for future expansion. If in the future, a faster method of transitioning to the hypervisor appears that requires more space, existing kernels can use it without modification.

Most of the other functions in the HVM control structure relate to the operation of the HVM domains. The store_cpu_guest_regs and load_cpu_guest_regs functions are used to completely save the state of a CPU. These save and load the state of the specified virtual CPU to the *Virtual Machine Control Block* (*VMCB*) or *Virtual Machine Control Structure* (*VMCS*) on AMD and Intel machines, respectively. For paravirtualized guests, Xen has to read each register in turn, and store it in a data structure. This is not ideal, because some registers can't be accessed in this way. HVM-capable CPUs generally have a single instruction that writes the entire CPU state to a pre-prepared area of memory. This region can then be stored and loaded later after a machine has been suspended.

Some of the guest's registers need to be accessed by other parts of Xen. These must be returned by the function that saves the state of the CPU. Two arguments are passed in, one for general registers and the other for control registers. If these are not NULL, then the store function should populate them with some registers that might be used externally (mainly the segment selector registers). The load functions must do this in reverse—copy the values from the passed functions into the control structure, from where they are then loaded en masse.

These functions both work on already extant VCPUs. For a paravirtualized guest, a virtual CPU is an entirely abstract concept; it's just a set of metadata that the hypervisor keeps around that is used to present the illusion of a real CPU to the guest. In an HVM environment, the VCPU is slightly more concrete; the CPU is aware of the existence of multiple virtual CPUs.

Initializing and destroying virtual CPUs for paravirtualized guests just requires creating the required structures within the hypervisor. For HVM guests, part of this initialization process is defined by the host CPU. The difference is somewhat analogous to the configuration of virtual memory on something like SPARC and x86. In one case, the page tables are only used by the operating system to populate the TLB. In the other, the hardware may read them directly without the operating system's intervention. The former case is similar to a pure PV guest's VCPU—the information is used to set the CPU state, but the CPU is not aware of the existence of the VCPU—whereas the latter is closer to the case for HVM guests, where the running CPU may directly access the VCPU's information.

The distinction is even more important on HVM-aware CPUs that include a tagged TLB (TTLB). In this case, contents of the TLB are not flushed between VM switches, but the VCPU ID is updated in the processor and only the TLB entries tagged with this ID are used when resolving virtual addresses. In this case, the physical CPU must be directly aware of the VCPU that is running.

Although the HVM support functions are defined in their own part of the tree,

they are used in the standard architecture-specific code, which itself is called from the platform independent part of the hypervisor. The `domain.c` file contains most of the functions that are used for setting up and manipulating a domain, and so is responsible for most calls to the HVM code.

When creating a new CPU, the vcpu_initialise () function is called. This, in turn, calls the vcpu_initialise () function in the hvm_funcs structure for HVM domains. The same delegation happens when destroying the VCPU or creating and destroying a domain.

HVM and Non-x86 CPUs

The term HVM is only used by Xen in the context of x86 CPUs. Other platforms supported by Xen may use the host platform's virtualization instructions, but they do not differentiate as strongly between hardware-assisted and software virtualization. This is caused by two things. Firstly, the situation found in the x86 world, where AMD and Intel both add incompatible extensions to the base instruction set, is relatively uncommon, meaning that there is less of a need for an abstraction layer to use chip-specific functions. The second reason is that most other platforms are capable of performing full virtualization even without extensions to the instructions set (although extensions may make it more efficient), removing the need to strongly differentiate between HVM and PV guests. When the Xen project began, HVM on x86 was not possible, so all guests were PV. It became necessary to introduce a new term to differentiate between the two categories.

At the time of writing, HVM technologies in x86 chips are still very young. The only (widely deployed) CPU line that has had hypervisor support for a relatively long time is IBM's POWER line. The high-end POWER systems have had their own hypervisor for some years, and are now supported by Xen.

The x86 HVM systems now allow unmodified systems to run, removing some of the original problems with x86 virtualization. These extensions are not yet, however, particularly fast. The current generation of HVM technologies from both AMD and Intel can be used to run unmodified operating systems, but provide little speed benefits. Existing pure-virtualization technologies, such as VMWare, have shown that a lot of tasks can be performed faster using binary rewriting than using the hardware assists.

Future generations of HVM-capable chips are expected to be significantly faster, making HVM assistance of paravirtualized guests attractive.

This page intentionally left blank

Chapter 14

Future Directions

Xen is under constant development by the team at XenSource and a large number of external contributors. It has evolved from a Ph.D. project to a successful commercial hypervisor supporting hardware assisted virtualization and paravirtualization on a number of platforms including x86, IA64, and PowerPC.

As a community-run open source project, the Xen project receives contributions from a large number of contributors—including research organizations and vendors. XenSource hosts the code repository, and Ian Pratt is the project leader; however, there are many submaintainers of different parts of the tree who take responsibility for their particular areas of expertise. Intel manages the VT subtree for example, and AMD manages the SVM code base. The project distinguishes between hypervisor core extensions (maintained by Keir Fraser) drivers (various maintainers), tools (Ewan Mellor) and the XenAPI, as well as the Linux kernel patches (Red Hat) and the CIM provider stack (Novell). There are also separate maintainers for each of IA64 (Alex Williamson HP) and PPC (Jimi Xenidis, IBM), 32-bit and 64-bit hypervisor support (Jan Beulich, Novell). The project leader ultimately reserves the right to admit any patches, but the shared maintainership roles ensure that there is no possibility that any one organization could control the code base.

14.1 Real to Virtual, and Back Again

One use for Xen is to run a legacy operating system "inside" a newer one. Although Xen, in fact, runs underneath both, the illusion that one OS is running as the guest of another is easy to maintain by exporting the user interface components from one guest to another.

Legacy operating systems typically have a large number of custom configu-

ration options set, and so reinstalling them in a virtualized environment is not always possible. In these instances, it is highly beneficial to be able to move them directly from their current location into a virtual domain. In some cases, it might be useful to be able to dual boot the legacy operating system, and either run it in native mode or in a virtual environment. A lot of work has been done with Xen to allow this. Windows, for example, can be dual-booted with XenoLinux, and either run natively or in an HVM guest. Because Xen does not currently support 3D acceleration of Windows guests, it can be used in a virtual environment for most legacy applications and then run natively when 3D support is required.

This facility can be used in conjunction with HVM-assisted virtualization as well, to allow an operating system to be installed as an HVM guest and then have the kernel replaced with a paravirtualization-aware version. This allows an operating system to be installed from original media without the need to reboot the computer without Xen for the install process.

The biggest problem with this approach is that installers boot in real mode, and often do some very messy things at the start. This is especially true of systems that boot from CD. On AMD systems, which permit virtualization of real mode guests, this is less of a problem; but on Intel systems, it requires emulation of much of real mode. Currently, the emulation code in Xen is undergoing a lot of work to make this process relatively painless.

14.2 Emulation and Virtualization

Virtualization can be seen as a special case of emulation, where the emulation function for the majority of operations is the identity function. Because it is already possible to migrate running machines from one Xen host to another, this leads to the question of whether it is possible to migrate one to a full emulator instead.

Conceptually, this is not difficult. During migration, the hypervisor is aware of the entire virtual machine state—register and memory contents as well as device states. This information can be used to set the emulator state and allow things to continue.

Running a virtual machine in emulation provides much finer-grained control over operations. An emulator can easily be run one instruction at a time, and in some cases even run backward. Input and output are strictly controlled, and this makes it an ideal platform for debugging.

Debugging is not limited to the software. If a virtual machine is experiencing problems at a particular point, it is possible to compare the results of the emulator to those of a real machine and see if there are discrepancies. Because it is often easier to verify the implementation of an instruction, or set of instructions, in an emulator than on a physical machine, this can be useful in debugging the hardware

as well. Of course, the bug may be in the emulator; again the capability to run a machine from the same state on both real and emulated hardware can help debugging here.

Beyond debugging, migration to emulated environments opens the door to cross-platform migration of virtual machines. A virtual machine running on a slow ARM chip in a mobile device could be migrated to a fast desktop or server machine. Even allowing for the overhead of emulation, this is likely to give a speed boost. If the guest kernel is aware of the migration, it can even take advantage of it directly. Code running in abstract machines such as the JVM or .NET CLR could have the JID cache invalidated and be recompiled from bytecode for the new native platform.

Currently, *virtual to emulated* (*V2E*) and *emulated to virtualized* (*E2V*) migrations using QEMU are still experimental. They are likely to become a significant feature of Xen in the future, however.

14.3 Porting Efforts

Porting means two things in the context of Xen. It can refer to porting the hypervisor to a new architecture, or porting other operating systems to run as Xen guests. New platforms are constantly being added, with all of the major CPU families now represented, either in the tree or by external porting efforts.

New operating systems are more interesting. Xen itself is more-or-less agnostic with respect to guests. It specifies a number of functions and interfaces that a Domain 0 guest must implement, but places very little constraint on unprivileged guests. For much of its development, Linux has been treated as the "standard" Domain 0 guest. This leads Xen to be regarded by many as a Linux virtualization system. This is not the case, however. NetBSD and Solaris both support running in Domain 0, and so there is no direct dependence on Linux.

Although Domain 0 can, in theory, be any operating system, in practice it is much easier to port UNIX-like operating systems. The existing management tools expect to communicate with the hypervisor via the `/dev/xen*` family of devices. A system that runs Python (the language in which the management tools are written) and exports devices as files is going to be easier than others to use as Domain 0.

These limitations are not present for unprivileged domains, however. These can do more-or-less anything that a native operating system can do. Writing device drivers for Xen is much easier than writing them for real hardware for two reasons. Firstly, only a single device driver needs to be written for each class of device. There is no requirement to write IDE drivers, SCSI drivers, SATA drivers, and so on, nor to support the various quirks of each individual controller. Instead, a domU guest simply needs to implement the block device driver. The same is

true of network interfaces. The second advantage is that the Xen devices are fairly abstract in design. The interface mirrors more the high-level description of how the device category should behave. This isolates developers from a lot of the low-level details of device driver design. As such, Xen makes an ideal platform for prototyping operating systems. Unlike emulated environments, the new system can run at near-native speeds, and still has access to a large variety of devices without the effort of directly supporting them all. By supporting protocols such as X11 and XGL, the guest can gain access to 3D acceleration without the need to support any 3D accelerators natively.

Xen recently had Plan 9 and Minix ported to it. Plan 9 is a research operating system, and so benefits greatly from having an abstraction layer between it and the hardware, allowing developers to concentrate on the novel aspects of the operating system, rather than supporting the wide range of commodity hardware available. Minix is primarily intended as a teaching tool, and can do this much better if students can run it concurrently with their other operating systems. Porting effort in this direction are likely to slow as HVM-compatible hardware becomes more common, as this allows guests to run without being aware of the hypervisor.

In terms of hardware support, Xen currently regards x86, PowerPC, and IA64 as stable platforms. PowerPC support is being added by IBM. Hypervisor support has been present in the POWER[1] series for quite some time. IBM, as discussed at the start of the book, was responsible for some of the earliest pioneers work in virtualization, and this has continued. High-end IBM hardware has included virtualization-aware devices for some time, and can be partitioned at the firmware level. Xen makes use of the same interfaces, but allows a finer-grained control over virtual machines.

Perhaps the most interesting platform under development is the ARM9 architecture. The port, by Samsung and others, is the most similar to the "original" Xen ideas because, unlike modern x86 and PowerPC chips, ARM9 has no native support for virtualization. It is a simple two-ring design, and an ideal target for paravirtualization. More importantly, ARM9 is an incredibly widely deployed architecture. A large number of PDAs and mobile phones use CPUs of this family, giving a lot of potentially interesting uses. Xen could be used for sand-boxing on these devices, allowing untrusted software to run in an unprivileged VM without affecting the rest of the system in the same way as Java, for example.

The current work on the ARM port makes use of a slightly different device model to "classic" Xen. Devices are divided into two categories—shared and exclusive—depending on how they are used. Things like mass storage devices (mass being a relative term in the context of a mobile device) and networks are

[1] Modern IBM POWER CPUs implement the PowerPC specification, with a few extensions, as do those branded as PowerPC. The difference between the two is now largely branding; POWER is aimed at high-end workstations and big iron, whereas PowerPC is aimed at cheaper workstations down to embedded devices.

viewed as shared devices, whereas the display and input devices are exclusive. Only one domain has access to the exclusive devices at any given time. A button on the device is typically used to switch between these. This provides an interface similar to virtual desktops to the user, although in this case each desktop is backed by a complete virtual machine.

14.4 The Desktop

Although Xen currently provides a very robust environment for server virtualization, support for the desktop leaves something to be desired. On the desktop, the market is largely owned by Microsoft Windows, and a large proportion of the potential market wants to run "Windows and something else," with the emphasis being on Windows. One of the biggest requirements for desktop virtualization is that Windows be able to run 3D applications, particularly games. 3D support is not particularly important on the server, but a modern desktop environment offloads a significant amount to the GPU even before you start running 3D applications.

With the inclusion of an IOMMU, this becomes quite simple, because Windows can run as an HVM guest and have the 3D card assigned to it. This is more difficult without an IOMMU, and unfortunately most desktops (and laptops) currently do not have one. One possible solution is to rearrange the memory layout so that a single HVM guest is at the bottom and top of the physical address space, with space reserved in the middle for the hypervisor and paravirtualized guests.

The disadvantage of this approach is that the HVM guest (Windows) can violate the protection mechanisms via DMA instructions sent to the hardware. This is unlikely to happen, because the emulated BIOS calls tell the guest that the portion of address space used by the hypervisor and other guests is reserved, and should not be used. It is possible, however, for a malicious program that has compromised ring 0 in the guest, or a buggy display driver, to break isolation. This is completely unacceptable in a server context, but it is not such a problem for a desktop, because a compromise to the Windows guest here is likely to be sufficiently serious that the additional loss from compromising the hypervisor is comparatively less important. This is not the case if the hypervisor is being used to sand-box potentially unsafe windows applications, or if the user stores important information on a non-Windows guest. In situations where full isolation is required, the user can disable the pass-through to the physical device from the HVM domain, or invest in a new system with an IOMMU.

A full IOMMU is not required to provide isolation here. The device exclusion vector on modern AMD chips provides protection, but not translation, for DMA, and so can be used in this context. The pseudo-physical to machine mapping for the HVM guest is simply the identity function, so no translation is required. The

DEV can be used to prevent the HVM guest from using the device's DMA controller to overwrite areas outside its own address space. This also has performance advantages, because the DEV is faster than a full IOMMU implementation. The main limitation of this approach is that it restricts a single HVM guest to the hardware access. In the desktop setting, this is likely to be acceptable; a user could run Windows for games and legacy applications and *NIX for real work, and switch between the two without needing a reboot. Alternatively, privileged domains could be used for security tools inspecting the HVM guest and verifying the absence of rootkits or other tampering.

The requirements for desktop virtualization are somewhat different from server virtualization. On the server, it is fairly common for virtual machines to be "owned" by different users. On the desktop, there is typically a single user, who wants to run multiple operating systems. The use may be security—running untrusted programs in a separate domain—or compatibility.

Because a single user is in charge, and physically present, that user is likely to be much more involved in management of the system. A user is likely to want to change the assignment of hardware to guests relatively frequently on a desktop. Although a server generally has a very static configuration, desktops (and laptops) often have external peripherals, such as scanners, mass storage devices, or even additional displays, plugged in and detached during operation. Beyond attachment, they might be delegated to different virtual machines at different times. One fairly common use case is to use the hypervisor as a substitute for a KVM switch and multiple physical machines. In this way, the keyboard, pointing device, and video would be switched between different domains by providing some kind of interrupt to the hypervisor.

The single-user nature of most desktops also impacts the scheduler design. A single user is likely to have her attention focused on a single virtual machine at any given time. This domain changes over time, and should receive a scheduling bonus, because any slowdown will be noticed a lot more than performance degradation of other domains.

The security aspect makes virtualization very attractive to the corporate desktop market. In a number of organizations, it is common to physically isolate the corporate network from the Internet, with the exception of a tightly controlled bridge for email. Dedicated Internet machines are provided for users needing external access. This drives up administration, power, space, and hardware costs. Virtualization provides a potential solution, where a second network card is installed in the secure machine, and delegated to an "insecure" virtual machine. This second network is then connected to the outside world. Even if the insecure VM is compromised, it can't leak information outside.

14.5 Power Management

One of the biggest limitations of Xen, to date, has been the lack of good support for power management. Supporting power management within Xen is relatively complicated. The simplest form of power management is the halt instruction, issued by an operating system when no process needs to run, which allows the CPU to enter a low-power state until the next interrupt is delivered. Even this is not available to Xen guests; they are not allowed to pause the entire CPU, because other guests are likely to need it. An equivalent operation, a "yield" scheduler operation, is provided instead. When all guests are idle, the hypervisor can then issue a halt instruction.

Power management is useful for all systems, but is essential for two categories of user:

- Mobile systems, such as laptops and PDAs
- High-density servers

These are two of the biggest consumers of virtualization. For mobile users, only needing to carry around a single laptop is a huge benefit. Going even smaller, virtualization provides the potential for much better security for devices such as mobile phones, a possibility being investigated actively by Samsung. Mobile devices are typically powered by batteries with a finite capacity, and so their usable time is inversely proportional to the amount of power consumed while in use.

For data centers, the situation is much worse. Last year, over 10% of the power consumption of the state of California was due to data center usage, and this equates to a significant proportion of the running costs. Virtualization is attractive for these installations, because it reduces the number of physical systems that are required, but this becomes less interesting if the individual systems consume more power.

Modern systems provide a much greater amount of flexibility in the realm of power management. CPUs from both AMD and Intel support frequency scaling. When the CPU load is below 100%, it is possible to reduce the clock frequency of the processor, resulting in much lower power consumption. Going beyond that, it is possible to suspend the entire system to disk or RAM or to shut off individual peripherals.

On x86 systems, most power saving functions are exposed via the *Advanced Configuration and Power Interface* (*ACPI*), an open standard published in 1996. ACPI is a mature standard, more than a decade old, and is very flexible. Unfortunately, although it is widely implemented and deployed, the standard of most implementations leaves something to be desired. Many vendors only test their implementations with Windows (and then, often only with a single version of Windows) and so miss a lot of bugs that cause problems for different ACPI drivers.

Much of the code in the Linux power management subsystem is there to try to work around these bugs in specific systems.

Adding ACPI support to the hypervisor causes a significant amount of code bloat. The obvious solution is to cheat in the same way that Xen does for other hardware support: delegate it to Domain 0. This is not trivial, however. Although Linux does have reasonable ACPI support, most of the triggers for using it are not applicable to Xen. Because virtual machines do not show up in the XenoLinux process table, the Domain 0 system is likely to believe that the machine is idle and put it into a low power state when another guest is busy.

To fully support power management, two things are required:

- A means for telling the Domain 0 guest to perform power saving actions
- A means for other guests to signal that power saving would not adversely affect them

The first is relatively simple; an event channel could be allocated to this use and set up by ACPI-aware Domain 0 guests when they boot. They could then be notified to perform power saving actions.

The second is more tricky. There are a number of ways in which power saving can affect a running guest. Frequency scaling is a problem already for some systems. AMD machines automatically reduce their clock speed in response to high temperatures, and this causes the TSC to advance at a slower rate. Because the TSC is used by domains to find the current wall-clock time, this causes guest clocks to get out of sync. To avoid this, the hypervisor must update the VCPU timekeeping structures every time the clock speed changes.

Transitions between power saving states take longer the deeper the low power state is. A pause-until-next-interrupt (C1 in ACPI terminology) state allows returning to fully operational mode immediately. At the opposite extreme, a suspend to disk mode can take several seconds to enter and return from. The hypervisor can detect when a domain is idle, but it is much harder to detect when a domain is going to suddenly de-idle. Entering a low power state just before a domain needs to perform a large amount of processing is decidedly suboptimal. A large amount of research is currently going on in this area outside the Xen community, and hopefully some results will be applicable to Xen.

In keeping with the Xen philosophy, responsibility for determining the correct power state should be delegated to the running guests. A system should be trusted to determine if entering a low power state would cause it problems. For HVM guests, this can be done by providing an emulated ACPI implementation. A guest can then set its suggested power state, and the hypervisor can use this to implement a power saving policy—for example, ensuring that it is in the highest power state requested by any guest, or a weighted average state. For paravirtualized guests, new hypercalls need to be added, or a split device with the back end

in Domain 0, which handles the power management. The latter approach seems more likely, because it keeps the hypervisor small.

Another concern when it comes to power management is akin to the scheduling problem. Power, like disk, RAM, and CPU usage, can be seen as a finite resource. In a mobile device, having a single domain able to flatten the battery would be a problem. One common use for virtualization in the context of mobile devices is security, and allowing a malicious program in a VM to flatten the battery, and thus provide an effective denial of service attack, would reduce this use significantly.

It is possible that future implementations will need to treat power as just another resource. The `PowerTOP` utility from Intel allows the power usage of various processes on a Linux machine to be determined, and something similar would be applicable to Xen. Each virtual machine could have a certain percentage of the system's power assigned to it, and have less runtime allotted to it as it approached this threshold. Tracking power usage within Xen is much harder than in Linux, however. CPU power is only a small part of the problem. When a virtual machine writes to the disk, it communicates to the back-end driver, which then performs the write. A naive approach would regard all of the power usage from spinning the disk as the responsibility of the domain containing the back end, and throttle it. This would slow performance of the entire system, without actually addressing the problem.

It is worth noting that there are two potential ways of measuring power in this context. For a mobile computing environment, it is energy, rather than power, that is the finite resource. Each domain could be granted a certain number of Watt-seconds, and be slowed and then stopped when it exceeds this. These allocations would then be reset after the battery was charged, and ignored while on mains power.

For fixed systems, there is no constraint on the total amount of energy available, but the power usage as a whole should be kept to a minimum. In a system where virtual machines are owned by different individuals or organizations, one user should not be allowed to use an unfair proportion of the power. In this case, the power assignments would need to be continuous, rather than discrete.

Power management support in Xen is likely to see a lot of changes in the next few years. XenEnterprise contains a full implementation of ACPI, which uses the Windows ACPI HAL, but equivalent support is not available in the open source version yet.

14.6 The Domain 0 Question

One of the biggest questions regarding the future of Xen is the place of Domain 0. A lot of functionality needs to be implemented by a guest wanting to exist in Domain 0. Initially, this was beneficial to Xen, because it meant that a lot

of code could be reused by Xen, without having to be directly imported into the hypervisor.

There are two major limitations to this approach, however. The first is that, as the responsibilities of Domain 0 grow, the difficulty of porting a new guest to Domain 0 grows. This means that we start to lose some of the flexibility of separating dom0 from the hypervisor; if it becomes too difficult to port other guests to dom0, Xen starts to depend fully on Linux (currently NetBSD and Solaris can also be run in Domain 0).

The other is that we start to lose out on security. A guest running in Domain 0 has a huge amount of access; it can map pages from other guests and poke around in their memory relatively easily. A compromise to the Domain 0 guest is likely to result in a compromise to the entire physical machine. This removes a lot of the advantage of separating Domain 0 out from Xen.

Two solutions have been proposed to this. The first is to import the required functionality from dom0 into Xen itself. This seems like a step backward, because the first version of the hypervisor did include a lot of this functionality, and eventually delegated it to Domain 0. This introduces some significant problems with maintenance, because the code would have to be forked (probably from Linux), and would need to be kept in sync with bug fixes from the source.

The second solution is to split the Domain 0 responsibilities. This is made a lot easier by the new security policy work, which allows access to privileged hypercalls to be restricted at a fine granularity. A lot of places in the Xen code include segments like this:

```
if ( current->domain->domain_id != 0 )
        return -EPERM;
```

If the caller of the hypercall is not Domain 0, they fail. These are all being gradually replaced with hooks into the security framework, which allow a policy to be loaded controlling which domains can access which hypercalls.

After this has been done, Xen begins to look more like a multiserver microkernel. The functionality of Domain 0 can be split roughly into two categories:

- Management, including creating and destroying virtual machines
- Hardware support, including providing the back-end drivers

The management component is currently quite dependent on Python. This is a problem, because it means that the Domain 0 kernel must be complicated enough to support Python, and that an extra 5MB of unaudited code (the Python runtime) must be included in Domain 0. There is some experimental work underway to reimplement these tools in OCaml, which can be compiled directly to native code and run from a very small kernel that does nothing other than manage the hypervisor.

Drivers can already be partially isolated using driver domains. The vast majority of code in any modern kernel is device drivers, and eliminating these from Domain 0 results in a much smaller amount of code to audit. For reliability, it is possible to keep each driver in a separate domain. This also has the advantage that different guests can be used to provide the drivers for different pieces of hardware; you can have Linux and NetBSD providing drivers, and select the operating system with the better support for each piece of hardware. If one crashes, it can simply be restarted in some cases.

14.7 Stub Domains

The current implementation of HVM support involves running emulated drivers in Domain 0. The most obvious problem with this is that it involves more code running in Domain 0, where it could potentially cause security issues. The slightly less obvious problem is that of scheduling.

Because the emulated devices are processes in Domain 0, their execution time is accounted to Domain 0. An HVM guest performing a lot of I/O can cause Domain 0 to use an inordinate amount of CPU time, preventing other guests from getting their fair share of the CPU.

The proposed solution to this is to use *stub domains*. These are small domains that run nothing other than the device emulators. Each HVM guest would have its own stub domain, responsible for its I/O. Starting a new HVM domain would involve creating two domains—the HVM domain itself and the stub domain—and the pair would then communicate with Domain 0 in the same way as a paravirtualized guest.

Although conceptually this is simple, it makes scheduling slightly more tricky. The current schedulers in Xen are based on the assumption that virtual machines are, for the most part, independent. If domain 2 is underscheduled, this doesn't have a negative effect on domain 3. This is not always true in the current case. Domains used to isolate servers in a producer-consumer relationship are one counterexample, and underscheduling of Domain 0 can affect all domains. With HVM and stub domain pairs, however, the situation is more pronounced. The HVM guest is likely to be performance-limited by the amount of time allocated to the stub domain. In cases where the stub domain is underscheduled, the HVM domain sits around waiting for I/O.

There are two potential solutions to this. One is to implement a mechanism similar to Doors, from the Spring operating system and later Solaris. Doors are an IPC mechanism that allows a process to delegate the rest of its scheduling quantum to another. If this mechanism were implemented for Xen, the HVM domain could be configured so that it was never directly scheduled. The stub domain would run whenever the pair needed to be scheduled. It would then

perform pending I/O emulation and, instead of performing a "yield" scheduler operation, it would perform a "delegate" operation on the HVM guest, which would then run for the remainder of the quantum. This same mechanism could be used for paravirtualized domains, to enable more accurate accounting of time spent performing I/O by a driver domain (or Domain 0) on behalf of the guest.

Another solution, proposed by IBM, is to introduce *scheduler domains*, based on work in the Nemesis Exokernel; these are similar conceptually to the $N : M$ threading model employed by some operating systems.[2] In this model, the hypervisor is responsible for scheduling groups of virtual machines, rather than individual ones. One domain would be defined as a scheduling domain. The hypervisor's scheduler would schedule this domain, and it would be responsible for dividing time amongst the others in the group. In this way, the scheduler domain fulfills the same rôle as the userspace component of an $N : M$ threading library.

Whichever approach is taken, stub domains are likely to be a big part of future versions of Xen.

14.8 New Devices

The current Xen release includes stable block and network devices, and newer virtual framebuffer and TPM devices. The biggest change to these is going to be the introduction of the new network device protocol (NetChannel2) described in Chapter 9. This will allow Xen guests to take advantage of network interfaces with hardware offloading capabilities better, and provide better performance for all network-related activities.

Support for sound devices is a logical next step for Xen, because they are found on most desktop and laptop systems and are still not supported. HVM guests can use sound via an emulated device that integrates with the mixer device in Domain 0. A paravirtualized sound device is currently under development. This will allow guests to read and write audio streams to an I/O ring and have them added to the mixer in Domain 0. Sound devices are quite interesting from the perspective of virtualization, because even relatively cheap ones have multiple hardware channels supported. This gives the potential for a virtualization-aware driver to give access to the ring buffers used by the hardware directly and only use "safe" mechanisms for controlling. This approach could allow very low-jitter audio output.

Although not, technically, a device, the *XenSocket* work provides the potential for very fast interdomain communication. XenSocket uses a shared memory transport between two domains to provide a faster means of connecting streams between domains than the network interface. Because XenSocket is purely for

[2] Formerly Solaris and FreeBSD, now NetBSD.

point-to-point connections between domains on the same machine, it can avoid all of the overhead of the protocol stack.

Performance of the XenSocket is closer to that of a pipe, or UNIX domain socket, than a network connection. This is likely to become more important, as a number of protocols already exist for sharing access to devices over a network and having a very fast and efficient network for interdomain communication would make it possible to use these from userspace in a lot of cases, rather than implementing a new device category at the kernel level.

Finally, better delegation of real hardware to guest domains is likely to become a priority for Xen as it aims more for the desktop, as is better support for hotplugging. The latter is very important for external peripherals, which are likely to be added and removed during a VM's lifecycle, and may want to be assigned to different domains on each insertion.

USB device support only works for guest domains by delegating the entire (PCI) USB controller to the domain, or by using a USB-over-IP protocol via the virtual network interface. This approach has a couple of problems. Linux is currently the only operating system with USB-over-IP support, restricting delegation of USB devices to systems with Linux in both Domain 0 and the unprivileged guest. The other problem is that encapsulating the USB protocol in Ethernet or IP involves an extra layer of overhead, at both encoding and decoding ends, that could be eliminated with a lighter protocol.

14.9 Unusual Architectures

The main focus for Xen is one-to-four socket machines. A couple of years ago, this meant one to four processors. Now, it generally means one to 16 cores, and is likely to increase to 32 cores at the top end fairly soon.

The original target for virtualization was big iron, which may have had a significantly larger number of processing units. These machines were used to consolidate independent minicomputers, with virtualization presenting virtual minicomputers to the users. Some people still want to use virtualization in this way.

With live migration and a storage area network, it is possible to run Xen on a cluster and do dynamic load balancing between the different machines, giving a similar effect. There is a class of machines, however, that fits somewhere between a cluster and a scaled-up desktop. The large scale *Non-Uniform Memory Architecture* (*NUMA*) machines sold by companies such as SGI fall into this group. Like low-end machines, they have a single address space, and can migrate processes between processors. Like clusters, the cost of accessing memory depends on the location.

These machines typically use a 64-bit physical address space, with a portion reserved for specifying which node owns the memory. This requires some extra

work on the memory allocator for the hypervisor. Domains should be allocated memory from the node on which they are running, in order to maintain the illusion of a uniform memory space to the running guests, and to allow sensible TLB interaction. Similarly, the scheduler needs to be aware of nodes, so that it doesn't attempt to run two VCPUs owned by a single domain on two different nodes.

The other big change comes from I/O device support. Although memory is shared between nodes, I/O space is not. A driver domain must be pinned to a single node, or it will suddenly find itself unable to access the hardware for which it is responsible. Even for fully paravirtualized guests, there can be problems caused by the fact that there some devices, such as the console, that exist on all nodes. If a guest is accessing the console on one node and is moved to another, problems are likely to arise.

This kind of issue is not likely to remain specific to big iron for very much longer. The current AMD designs have a memory controller on each core. Accessing memory that is controlled directly by the local core is cheaper than accessing memory owned by another one. The difference is much smaller than the difference between memory and in local and remote machines in a large NUMA system such as an Altix, but it does exist.

Current Intel chips also exhibit some of these properties, because the level two cache is shared between cores on the same die, but not cores in different sockets. This means that sharing memory between VCPUs scheduled on cores in the same socket is going to be cheaper.

As this trend continues, it is likely that operating systems will begin to be aware of the NUMA nature of their underlying platforms and take advantage of it for performance. The hypervisor, if it wants to remain competitive, will need to support these NUMA-aware guests as well as ones that prefer to pretend that memory is uniform.

While large NUMA systems are relatively rare, they are still fairly similar to existing configurations that are well supported by Xen. The biggest change that is likely to alter this is heterogeneous multicore systems.

A modern PC can be seen as a heterogeneous multicore system already, because it includes one or more general purpose processors and a dedicated parallel stream processor in the form of the GPU. Typically, the GPU is controlled via a special-purpose API, such as OpenGL or DirectX. This API is responsible for multiplexing access to the device, typically with some input from the windowing system.

In the past, the trend has been a cycle between general and special purpose hardware. Special purpose hardware has been popular for a little while, until the general purpose CPU becomes "fast enough," at which point it becomes cheaper to just use the general purpose execution units. Examples of this include modems and sound processors.

This cycle is likely to change somewhat, because power is becoming increasingly important. Although it is cheaper to use the CPU for everything (because

you need fewer processors), it is not more power-efficient to use general purpose processors for everything. Although GPU functions are likely to be subsumed by CPUs in the next few years, this will only happen by making the CPU more GPU-like—adding stream-processing extensions to existing instruction sets. Both AMD and Intel appear to be heading in this direction.

Beyond the GPU, other special purpose processors that are exposed to userspace programs include cryptographic coprocessors. These are also typically interacted with via a library, such as OpenSSL. As with GPUs, CPUs are beginning to absorb some of these functions. Chips from manufacturers such as PA Semi and VIA include cryptographic acceleration.

The best example of a heterogeneous multicore system at the moment is the Cell processor. This includes a single PowerPC core, which could potentially run the existing PowerPC Xen port when support for PowerPC systems without hypervisor extensions is finished. The remaining seven or eight cores,[3] known as *Synergistic Processing Units* (*SPUs*) are highly specialized vector processors with their own local memory and DMA capabilities. Embedded processors are also tending toward a heterogeneous model, with many containing a general purpose ARM core and a number of specialized units, such as DSPs.

How these are to be exposed to guest domains is an open question. Highly specialized execution units could be exposed as simple devices. More general ones are likely to be exposed in a manner closer to current CPUs. Future versions of Xen may need to associate an instruction set with each VCPU, and allow guests to schedule tasks on VCPUs of different types, which are then scheduled on heterogeneous cores. Scheduling for these would have to happen in a different way to existing systems, because context switching is typically much more expensive, resulting in a need for scheduler with different-sized quanta for different VCPU types. Dependencies between components running on different processor types further complicates scheduling.

This ties in with the work being performed by Samsung on partial relocation, where only some of the VCPUs of a running guest are migrated. Potentially, this could be extended to allow a VM to own VCPUs running on CPUs on machines with different architectures. A guest that ran on an ARM system might bring up an x86 or PowerPC VCPU on a different machine to handle a task involving a number of floating point operations.

14.10 The Big Picture

The current hypervisor is close to the Xen vision of a thin abstraction layer for virtual machines. The biggest change to the Xen landscape recently has come

[3]The Cell is designed with eight, but the mass-market version only uses four to enable chips with flaws to be used.

from HVM support in modern x86 hardware. This reduces the need for paravirtualization, to a degree, and is likely to be used to improve the performance of even PV guests. The changes between versions 2 and 3 of Xen were much smaller than the changes between versions 1 and 2. Version 2 moved a lot of functionality from the hypervisor into Domain 0, whereas the biggest change moving to Xen 3 was the addition of the XenStore. Most other changes were only visible from Domain 0, such as interrupt routing.

Future changes for Xen are likely to be incremental, for a while. Support for more devices is an obvious improvement, both by adding new categories of virtual device and better support for delegating physical devices to guests. The latter is expected to improve dramatically as IOMMUs become mainstream.

HVM support is still relatively young, and will continue to evolve as the hardware develops and is better understood. The distinction between HVM and PV guests is likely to blur from both directions. The domU XenoLinux prototypes from Intel show how an HVM guest can move toward paravirtualization by booting in HVM mode and gradually replacing its own code with hypervisor-aware versions. In the other direction, PV domains are likely to become hardware assisted to a greater degree. Which approach a guest operating system favors is likely to be defined by its age. A new operating system can use Xen as a hardware abstraction layer better by using the PV interfaces and using the hardware acceleration if the CPU and hypervisor support it. An existing operating system that already works on x86 can take advantage of PV-on-HVM capabilities to gradually support Xen.

The hypervisor has had a lot of change in its schedulers over the last few releases, with a scheduling algorithm generally not lasting longer than a major release. The Credit Scheduler overcomes most of the limitations of earlier approaches, and is likely to see minor improvements to address new challenges, such as NUMA systems and heterogeneous cores.

Domain 0 is likely to be reduced in importance, being replaced by individual driver domains and a management domain. The management domain will either be an existing Domain 0 operating system, or possible something new, designed for Xen. The functionality of the management domain may even be split further, giving a small cluster of domains with responsibility for different management tasks.

This devolution from Domain 0 is made possible by the new security framework being added to Xen. This also provides a means of restricting the activities of unprivileged domains. The construction of policies that provide good security without providing too much of a barrier to usability is likely to be a significant research area within Xen in the near future.

A lot of effort has recently gone into the administration tools for Xen. One of the biggest differences between Open Source Xen and the commercial offerings from XenSource is the ease of administration. The hooks required for more advanced tools are gradually making their way into the Open Source code base, in

the form of the Xen API. A number of tools are being built on top of these by the community.

The typical software development cycle can be categorized as "make it work, then make it fast." Xen is coming to the end of a "make it work" phase, and is likely to enter a period of stabilization and optimization for a little while. This is not to say new features will not be added; they will, but the primary focus is likely to be on filling in gaps rather than on major new additions. By virtue of the approach used, Xen had a significant performance advantage over competing virtualization systems from the start. The comparison now is not between Xen and other hypervisors, but between Xen and native hardware. Any area in which performance lags significantly behind native speeds needs addressing.

This page intentionally left blank

Part IV

Appendix

This page intentionally left blank

Appendix

PV Guest Porting Cheat Sheet

Porting an operating system to Xen is very similar to porting it to any other new platform. If the system already runs on x86, the number of required changes is relatively small compared to porting to a completely new architecture. At the simplest level, you do not need to switch to a new compiler, because your current one must already be able to generate x86 binaries. You also do not need to worry about alignment or endian issues, because these are exactly the same for native and paravirtualized x86.

Most of the changes are likely to be at the lowest level. If, like NetBSD, you already have an abstraction layer for things like interrupts and memory management, you only need to modify the abstraction layer to support running in an unprivileged domain. The biggest change is likely to come from timekeeping. A paravirtualized guest system has to understand that it is sharing the CPU, and so CPU time and wall clock time are not necessarily interchangeable. This might require some significant changes, because most operating systems that were not designed to be run in virtualization are unlikely to already support this abstraction. This chapter will provide a quick recap of all of the changes that need to be made when porting an operating system to Xen.

A.1 Domain Builder

The domain builder is responsible for configuring the initial layout of a new domain. If your boot process is similar to Linux—an ELF kernel binary in a flat, paged, address space—you can likely use the Linux builder. If not, you may need to write your own.

Minix 3 is an example of an operating system that required a new domain builder. Unlike Linux and modern BSDs, the Minix kernel binary is in a.out format, not ELF. Plan 9 and Minix both run on Xen and use a.out format kernels, so either of their domain builders could be used as a starting point when writing one.

Another key difference is that Minix uses segmentation, rather than paging, for memory protection. By default, Xen assumes that guest kernels want to have a flat memory layout and use paging for memory protection. To facilitate this, it installs a GDT, which provides entries for flat kernel and userspace address spaces. If you use segmentation, you need to configure your own GDT, either in the domain builder of at the start of the kernel boot process.

Adding a new domain builder involves modifying the Xen code. This is a problem, because it means that you cannot boot your guest on unmodified Xen. Although this may be an acceptable solution in some situations, such as for testing or limited deployment, it is not ideal. A better solution is to create a boot loader that can use an existing domain builder, and then bootstrap the kernel.

A.2 Boot Environment

After booting, a guest kernel may not call the BIOS. Instead, it needs to inspect the start and shared info pages to get the amount of free memory and the available processors.

One of the first things that needs to be written is a driver for the XenStore. Only the console and XenStore devices can be constructed from information in the start info page. The remainder need to be bootstrapped using the store.

The XenStore is a fairly simple device, which closely mirrors the console in design. After a driver for one is written, adding a driver for the other is relatively simple. A console driver allows some basic user interaction, and so it makes development much easier, although later in the boot process something like the virtual framebuffer device should replace it if more involved user interaction is required.

A.3 Setting Up the Virtual IDT

Setting up the IDT is slightly different on Xen to native x86, because the way interrupts are delivered depends on their type. Exceptions can be delivered to a function specified by the interrupt vector as per usual. These handlers are installed via the trap table hypercall. Setting these up is fairly simple, and mirrors the installation of the IDT, although the structure of the trap table is slightly different.

Other interrupts caused by physical devices are less important when porting to domU, but become more so when you begin to add Domain 0 support. Because these can occur when the VM is not scheduled, and thus another guest's IDT is installed, they must be mapped to event channels. Installing the callback for event delivery is one of the first things a guest must do on boot. After this, it should individually mask all event channels and then enable event delivery. The channels for the XenStore and console devices are likely to need to be unmasked shortly after, and then others as the associated drivers are brought online.

If you intend to do preemptive multitasking, or offer any clock-based services, you are likely to want to set up the clock virtual device as well. This raises a virtual IRQ, which must be mapped to an event channel. The clock VIRQ is raised periodically in domain virtual time, and so it is important for the kernel to have a mechanism for differentiating between this and real (wall clock) time. A new kernel, designed with virtualization in mind, is likely to have this abstraction from the start, but a ported kernel is likely to be built on the assumption that the two are the same and so some major modification might be needed.

A.4 Page Table Management

When porting a guest to Xen, it is a good idea to start by using writable page tables. In this mode, only the page directory must be manipulated via hypercalls; the rest of the page tables can be modified directly. This makes it much easier to port operating systems that use paging.

After the new guest is working in this mode, it is likely that a performance increase can be gained by moving to using fully paravirtualized memory management. To begin with, you should stop directly manipulating the page tables and use the MMU update family of hypercalls. These can be used even when page tables are writable. This allows you to gradually replace parts of the memory management code, and check that things still work. After these are all replaced, you can turn off the writable page table assist, and hope you didn't miss anything. At this stage, attempting to write to the page tables causes a fault, which you can catch and use to track down the offending code.

Because most modern processors don't support segmentation, it is relatively uncommon for an operating system to use it. Some, which are not intended to be portable to non-x86 architectures, do; these need some additional work in porting. As mentioned earlier, the boot loader or domain builder typically needs to be modified to present an initial environment. The initial GDT for such systems is usually created before the kernel starts. This is problematic, because the bootloader that does this expects to start in real mode, and requires extensive modification to start in protected mode. The LDT on such operating systems can be installed as on native hardware, but the GDT must be installed

using the HYPERVISOR_set_gdt() hypercall. This takes two arguments, a list of up to 14 page frames containing the GDT and a count of the number of entries in the new GDT. The hypervisor validates these before installing them. Note that a paravirtualized guest has fewer available GDT entries than a native kernel, because the hypervisor uses some of these entries.

A.5 Drivers

At a minimum, it is likely that a newly ported guest needs to support the console, XenStore, and block devices. The first two of these have similar interfaces, described in earlier chapters.

For debugging, the emergency console can be used. This is enabled for Domain 0 at all times, and for other domains if the hypervisor was built in verbose mode (currently, this is set when it is built with `make debug=y`). The console is connected to the first serial port in the system by default, but can be redirected passing the `console=` argument to the hypervisor at boot time. Setting it to `vga` will cause debugging output to be sent to the screen.

Because the debugging console is not usually available, and the standard console is, it is a good idea to switch as soon as you have mapped the page containing the console ring device, and only return to it if you have memory errors that are damaging this mapping.

The block and network interfaces have similar designs, and so are likely to be added as a pair, as long as the kernel has a network stack. These devices use the standard request-response ring mechanism in shared memory and an event channel to notify the two ends of activity on the ring.

A guest may want to implement other device categories later, such as the virtual TPM or framebuffer. These are likely to come much later in the boot process, because on most systems they are not prerequisites for a successful boot.

A.6 Domain 0 Responsibilities

The existing Xen tools are all written in Python, and expect a UNIX-like environment. This makes it very easy to allow other UNIX-like systems to run as Domain 0, but much harder for other systems. If your platform already supports Python, it is likely to be relatively easy to add Domain 0 support to an operating system that already runs in domain U. If not, you also need to create your own management interface, which is likely to be a lot of effort. The addition of the Xen API makes this slightly easier, because a platform that has some mechanism for providing an XML-RPC server could implement a `xend` replacement and run `xm` remotely (or in a domU guest).

Domain 0 is expected to provide the back ends for paravirtualized devices and the XenStore, as well as perform system management tasks such as creating and destroying domains. The minimalist design of the hypervisor adds a significant burden to guests running in Domain 0. This is intentional, because it reduces the code running in ring 0, but makes adding Domain 0 support a significant task. It is recommended that an operating system be first ported to run as domain U, and thoroughly tested in this setting, before Domain 0 support is considered. In addition to simply hosting the back end devices, the Domain 0 guest is responsible for handling all of the multiplexing of device I/O from multiple virtual devices into a single physical one.

In future versions of Xen, it is likely that the responsibilities of Domain 0 will be decomposed, allowing Domain 0 support to be added piecemeal. Driver domains currently provide an example of this. A domain U kernel can be modified to run as a driver domain, accessing hardware directly, but not providing any management facilities. In the future, individual management functions are likely to be used to split between multiple domains, for added security, and then a domain will be able to start adding support for traditional Domain 0 features one at a time.

A.7 Efficiency

The main reason to use paravirtualization over full virtualization is performance. If this is your goal, you should try to ensure that your kernel is as efficient as possible. When writing userspace programs, it is often possible to get a speed improvement by using things like readv and lio_listio instead of large numbers of simple calls. This is because system calls are relatively expensive operations. The same is true of hypercalls.

If you are updating the page tables, you can group a number of updates together and issue a single hypercall. Something similar can be done in the general case. The HYPERVISOR_multicall hypercall can be used to issue a group of unrelated hypercalls. Using this means that the CPU performs the transition from guest to hypervisor (ring 1 or 3 to 0) only once for the complete set of hypercalls.

Listing A.1: Multicall argument structure [from: xen/include/public/xen.h]

```
 */
struct multicall_entry {
    unsigned long op, result;
    unsigned long args[6];
};
```

Listing A.1 shows the format for elements of the array pointed to by the first argument to this hypercall. The second is the number of elements in the array. The op field contains the hypercall number; the value that would go in

EAX for a single call. The arguments, likewise, are moved from abx and so on to the corresponding fields in the structure. One thing to note is that the result is stored in a separate field, rather than clobbering the hypercall number. It is also important to remember that, unlike individual hypercalls, a guest can be preempted in the middle of a multicall, and may need to manually restart it.

A.8 Summary

Porting an x86 kernel to Xen is a simpler task than porting it to something like PowerPC, but it should still be viewed as a porting effort. Memory management is different, as is hardware support. Even the equivalent of the boot firmware is different to a native x86 host.

A lot of code can be reused. Building the page tables is almost the same on Xen/x86 as it is on native x86, with the only difference being the way in that they are installed (via a hypercall, rather than directly). A lot of the required changes can be implemented as macros, allowing the Xen port to be compiled from the same source files as the original.

Hardware support on Xen is very easy for domain U guests. Xen acts as a hardware abstraction layer, so only one driver is required for each category of device. This abstraction makes Xen an attractive target for operating systems with limited developer resources, including new and experimental systems which have never run on real hardware.

Index

3D support, 257

8086, 27

ACPI, *see* Advanced Configuration and Power Interface
Adding new devices, 187
Address Space ID, 238
Administration tools, 200
Advanced Configuration and Power Interface, 259
Advertising devices, 187
AMD-V, 13
ASID, *see* Address Space ID
Asynchronous notification, 119
Atropos scheduler, 219

Behavior sensitive instructions, 4
Binary rewriting, 10
BIOS, 47, 244
blkif, *see* Virtual block device
blkif_front_ring_t, 163
blkif_request_segment, 165
blkif_request_t, 165
blkif_sring_t, 163
Block cache, 166
Block device, 36
    connecting, 163
    initializing, 162
    loading data, 167
    storing data, 165
    supporting CDs, 177
    XenStore nodes, 162
Boot firmware, 47
Boot trampoline, 40
Booting, 27
Borrowed virtual time scheduler, 219
Breakpoints, 10

C bindings, 200
Cache flushing, 92
Calling convention, 30
Calling convention, hypercall, 12
CD drives, 177
CIM, *see* Common Information Model
CIM-XML, 210
Common Information Model, 209
Compartmentalization, 7
Console, 49
Console device driver, 112
Console driver, 133
Console interrupt, 123
Context switch, 4
Control sensitive instructions, 4
Core devices, 161
CPU architectures, 256
CPU Virtualization, 4
CPUID in HVM mode, 236
Credit scheduler, 219, 222
Cryptographic coprocessors, 267

DEC Alpha, 4
Desktop Xen, 257
DEV, *see* Device Exclusion Vector
Device drivers, 99
    block device, 161

console, 112
framebuffer, 178
network interface, 169
PCI, 184
TPM, 183
USB, 186
XenStore, 150
Device Exclusion Vector, 14, 239
Device multiplexing, 100
Device scheduling, 224
Direct Memory Access, 6
Distributed Management Task Force, 209
DMA, 6
DMTF, *see* Distributed Management Task Force
dom0, *see* Domain 0
Domain, 19
Domain 0, 19
Domain 0 devolution, 262
Domain builder, 273
Domain U, 19
Domain virtual time, 53
domctl hypercall, 228
domU, *see* Domain U, *see* unprivileged domain
DOS, 9
Driver domains, 102

E2V, *see* Emulated to virtual
Earliest deadline first, 221
EDF, *see* Earliest deadline first
EFI, *see* Extended Firmware Interface
Emulated devices, 245
Emulated to virtual, 255
Emulation, 3
Event channel
bitfields, 51
Event channels
assigning to a VCPU, 127
binding, 125
closing, 129
masking, 130
polling, 133
querying status, 129
signalling, 128
Event ports, 33
Event trampoline, 134
Event types, 123
Event upcall, 137
event_channel_op hypercall, 124
Events, 111, 119
Extended Firmware Interface, 18

Firmware, boot, 47
Flush TLB, 92
Flushing cache, 92

GART, *see* Graphics Address Remapping Table
GDT, *see* Global descriptor table
gettimeofday(), 54
Global descriptor table, 76
GMFN, *see* Guest machine frame number
gnttab_copy_t, 64
gnttab_map_grant_ref_t, 62
gnttab_transfer_t, 64
GPFN, *see* Guest page frame number
Grant reference, 35, 61
Grant table operations, 66
Grant tables, 34
grant_entry_t, 68
grant_table_op hypercall, 61
Graphics Address Remapping Table, 7
Guest loader, 39
Guest machine frame number, 79
Guest page frame number, 79

Handling events, 134
Hardware assisted
device pass-through, 239
DMA safety, 239
page tables, 238

real mode, 239
virtual CPUs, 238
virtual interrupt routing, 240
Hardware page tables, 75
Hardware virtual machine, 22, 23, 29, 235
Heterogeneous multicore, 266
Hibernate, 260
HVM, *see* Hardware virtual machine, *see* Hardware virtual machine
HVM hypercalls, 236
hvm_function_table, 248
hvm_op hypercall, 246
Hybrid virtualization, 14, 240
Hypcall calling convention, 12
Hypercall, 11, 30
Hypercall API, 197
Hypercall macro, 30
Hypercall page, 12, 30
Hypercall page setup, 248
Hypercalls
domctl, 228
event_channel_op, 124
grant_table_op, 61
hvm_op, 246
memory_op, 84, 93
mmu_update, 89
mmuext_op, 91
multicall, 277
sched_op, 132
set_gdt, 93
update_va_mapping, 90
update_va_mapping_otherdomain, 90
vm_assist, 83
Hypercalls, HVM, 236, 243
Hypervisor-based copy, 64

I/O rings, 65, 103, 164, 188
I/O virtualization, 36
IDT, *see* Interrupt descriptor table
Infiniband, 175
Input/Output Memory Management Unit, 6, 102, 239
Interdomain communication, 59
Interdomain events, 123
Interprocess communication, 34, 59
Interprocessor interrupts, 124
Interrupt 80h, 11, 120
Interrupt 82h, 30
Interrupt descriptor table, 94, 120
Interrupt handlers, 274
Interrupt vector, 28
Interrupts, 111, 119
Intradomain events, 124
Invalidate TLB entry, 92
IOMMU, *see* Input/Output Memory Management Unit
IPC, *see* interprocess communication
IPIs, *see* Interprocessor interrupts
IVT, *see* Virtualization Technology for x86

Jumbo frames, 173

Kernel header, 39
KQEMU, 236

Laptops, 259
LDT, *see* Local descriptor table
libcurl, 205
libvirt, 200
libxen, 201
libxml, 206
Local descriptor table, 76
Lockless ring buffer, 103

Mach ports, 33
Machine frame number, 79
Mapping memory, 61
Mapping the XenStore, 150
Masking events, 130
memory, 59
Memory assists, 82
Memory barriers, 107

Memory Management Unit, 5, 75, 238
Memory model, 75
Memory model, Xen, 78
Memory pages
    copying, 34, 64, 174
    exchanging, 86
    granting access, 66
    mapping, 61
    protecting, 77
    sharing, 34
    transferring, 34, 63, 174
Memory protection, 77
memory_op hypercall, 84, 93
Message passing, 34
MFN, *see* Machine frame number
Migration, 94
Minix, 256
mmap(), 60
MMU, *see* Memory Management Unit
mmu_update hypercall, 89
mmuext_op, 91
mmuext_op hypercall, 91
Mobile systems, 259
Model-specific register, 243
Mouse tracking, 181
MSR, *see* model-specific register
multicall hypercall, 277
multicall_entry, 277
MULTICS, 28
Multitasking, 7, 217

Native device drivers, 184
Nemesis Exokernel, 264
Nested Page Tables, 14, 238
NetBSD, 69
NetChannel2, 174
netif, *see* Virtual network interface
netif_extra_info, 172
netif_rx_response_t, 173
netif_tx_request_t, 171
Network device
    initializing, 169
Network interface
    receiving, 173
    transmitting, 171
New devices, adding, 187
Non-Uniform Memory Architecture, 265
Nonlocal I/O, 266
NPT, *see* Nested Page Tables
NUMA, *see* Non-Uniform Memory Architecture

Operating system support, 255

P2V, *see* Physical to virtual
Pacifica, 13
Page directory, 78
Page directory base register, 78
Page directory entry, 78
Page fault handling, 94
Page faults, 241
Page frame number, 79
Page table entry, 78
Page table management, 275
Page table updates, 89
Page tables, 78
Page tables, nested, 14
Page tables, shadow, 14, 82
Page tables, writable, 82
Paged real mode, 239
PALCode, 4
Paravirtual I/O on HVM, 246
Paravirtualization, 10
Partial relocation, 267
PCI devices, 184
PDAs, 259
PDE, *see* Page directory entry
Physical IRQs, 123
Physical to virtual, 253
Pin page table entry, 92
Plan 9, 256
Platform PCI device, 23
Popek and Goldberg, 4

Porting to Xen, 273
Porting Xen, 255
Power management, 259
PowerTOP, 261
Privilege rings, 28
Privileged instructions, 4, 28
Privileged operation, 30
Protected memory, 78
Protected mode, 17
PTE, *see* Page table entry
Python bindings, 200
Python tools, 207
pyxen, 201

QEMU, 236
QEMU Accelerator, 236

RDP, 178, *see* Remote Display Protocol
Real mode, 17, 27
Realtime scheduling, 220
Remote attestation, 183
Remote Display Protocol, 182
Requesting events, 124
Resume, 94
Ring buffers, 36, 65, 103, 164, 188
ring.h, 65
Round robin scheduler, 219

S/360, 8
sched_op hypercall, 132
Scheduler
  adding, 229
  adding domains, 225
  adding virtual CPUs, 225
  API, 218
  Atropos, 220
  borrowed virtual time, 219
  configuring, 228
  credit scheduler, 222
  defining, 224
  hypercall interaction, 228
  initialising CPUs, 225
  interface, 218
  realtime, 220
  running, 225
  Simple EDF, 221
  SMP support, 225
  stub domains, 263
  work conserving, 221
scheduler, 218
Scheduler domains, 264
Scheduler operations, 132
Secure Virtual Machine, 237
SEDF, *see* Simple EDF scheduler
Segment registers, 76
Segmentation offload, 171
Segmented memory, 80
Sending events, 128
Sensitive instructions, 4
set_gdt hypercall, 93
Shadow page tables, 14, 82, 238
Shared info page, 95
Shared memory, 34, 59
Shared memory buffers, 103
shared_info_t, 51
shmget(), 60
Signals, 33
Simple EDF scheduler, 219
SimpleKernel, 157
Simplest kernel, 38
Single user virtualization, 258
Sleep, 260
Solarflare, 175
SPARC, 6
Split driver model, 35
Split drivers, 100
SPUs, *see* Synergistic Processing Units
Start info page, 47
start_info_t, 48
Strongly ordered CPU, 107
Stub domain scheduling, 263
Stub domains, 245, 263
Suspend, 94, 260
SVM, *see* Secure Virtual Machine

SVPC WG, *see* System Virtualization, Partitioning, and Clustering Working Group
Synergistic Processing Units, 267
System call, 11, 30, 120
System Virtualization, Partitioning, and Clustering Working Group, 209
System/360, 8

Tagged translation lookaside buffer, 238
task_slice, 224
Threading models, 217
Time keeping, 53
Time-Stamp Counter, 53
Timer device, 123
Timer interrupt, 123
TLB, *see* translation lookaside buffer
TPM, *see* Trusted Platform Module
Transferring memory, 63
Translation lookaside buffer, 75, 90, 238
Trap table, 94, 120
trap_infoi_t, 121
Traps, 120
Trusted Platform Module, 177, 183
TSC, *see* Time-Stamp Counter
TTLB, *see* tagged translation lookaside buffer

Unmodified guests, 235
Unprivileged domain, 22
update_va_mapping hypercall, 90
update_va_mapping_otherdomain hypercall, 90
Updating the virtual framebuffer, 180
USB, 186
USB-over-IP, 187
Userspace network drivers, 175
Userspace tools, 200

V2E, *see* Virtual to emulated
V2P, *see* Virtual to physical
VCPU, *see* Virtual CPU
VGA BIOS, 244
VIRQ, *see* Virtual IRQs
Virtual 8086, 9
Virtual appliances, 8
Virtual block device, 161
Virtual CPU, 51, 119, 220
Virtual device bus, 109
Virtual devices, 99, 100
Virtual disk, 161
Virtual framebuffer
  updating, 180
Virtual IRQs, 123
Virtual keyboard, 180
Virtual Machine Control Block, 250
Virtual Machine Control Structure, 250
Virtual machine lifecycle, 37
Virtual memory, 78
Virtual network interface, 169
Virtual pointing device, 180
Virtual servers, 7
Virtual sound devices, 264
Virtual time, 53
Virtual to emulated, 255
Virtual to Physical, 253
Virtual-Processor Identifier, 238
Virtualization
  CPU, 4, 217, 220
  full, 235
  hardware-assisted, 235
  hybrid, 240
  I/O, 5, 36, 99, 224, 235, 239
  RAM, 4, 78
Virtualization Technology for devices, 240
Virtualization Technology for x86, 237
Virtualization, hybrid, 14
VM/370, 8
vm_assist hypercall, 83
VMCB, *see* Virtual Machine Control Block

VMCS, *see* Virtual Machine Control Structure
VMS, 28
VMWare, 10
VMX, 13
VNC, 178
VPID, 238
VT, *see* Virtualization Technology for x86
VT-d, *see* Virtualization Technology for devices
VT-x, *see* Virtualization Technology for x86

Wall clock time, 53
Weakly ordered CPU, 108
Windows guests, 235
Work conserving schedulers, 221
Writable page tables, 82
WS-Management, 210

X11, 178
x86 memory model, 75
x86 page directory, 78
x86 privilege rings, 28
x86 segmentation, 80
Xen API
  C bindings, 203
  console, 203
  host, 201
  host CPU, 202
  metrics, 202
  physical block device, 202
  physical network interface, 202
  session, 201
  virtual block device, 203
  virtual machine, 202
  virtual network interface, 203
  virtual TPM, 203
Xen API, 197
Xen API classes, 201
Xen dæmon, 197, 206
Xen device drivers, 100
Xen driver model, 35
Xen event model, 33
Xen interface hierarchy, 200
Xen maintainers, 253
Xen management API, 197
Xen master, 208
Xen master (command line tool), 197
Xen memory layout, 80
Xen memory model, 78
Xen networking, 169
Xen security, 262
xen_domctl_scheduler_op, 228
xen_machphys_mapping_t, 87
xen_machphys_mfn_list_t, 86
xen_memory_exchange_t, 86
xen_memory_reservation_t, 84
xen_pci_op, 185
xen_pci_sharedinfo, 184
xen_translate_gpfn_list_t, 88
XenBus, 109, 142, 163
xenbus_device, 109
xencons_interface, 113
xenfb, *see* Virtual framebuffer
xenfb_page, 178
xenfb_update, 180
xenkbd_key, 181
xenkbd_page, 181
xenkbd_position, 183
XenSocket, 264
XenStore, 36, 187
  device structure, 145
  mapping, 150
  message structure, 146
  message types, 147
  reading a response, 153
  userspace tools, 148
  writing a message, 152
XenStore device, 145
XenStore hierarchy, 142
XenStore interface, 141
xenstore_domain_interface, 145

xm, *see* Xen master
XML-RPC, 198
XML-RPC Data Types, 198
xsd_sockmsg, 146, 151

z/VM, 8