

OPERATING SYSTEMS IN DEPTH



THOMAS W. DOEPPNER

This page intentionally left blank

Operating Systems in Depth

This page intentionally left blank

OPERATING SYSTEMS IN DEPTH

Thomas W. Doeppner
Brown University




WILEY

JOHN WILEY & SONS, INC.

vice-president & executive publisher	Donald Fowley
executive editor	Beth Lang Golub
executive marketing manager	Christopher Ruel
production editor	Barbara Russiello
editorial program assistant	Mike Berlin
senior marketing assistant	Diana Smith
executive media editor	Thomas Kulesa
cover design	Wendy Lai
cover photo	Thomas W. Doeppner

Cover photo is of Banggai Cardinalfish (*Pterapogon kauderni*), taken in the Lembah Strait, North Sulawesi, Indonesia.

This book was set in 10/12 Times Roman. The book was composed by MPS Limited, A Macmillan Company and printed and bound by Hamilton Printing Company.

This book is printed on acid free paper. 

Founded in 1807, John Wiley & Sons, Inc. has been a valued source of knowledge and understanding for more than 200 years, helping people around the world meet their needs and fulfill their aspirations. Our company is built on a foundation of principles that include responsibility to the communities we serve and where we live and work. In 2008, we launched a Corporate Citizenship Initiative, a global effort to address the environmental, social, economic, and ethical challenges we face in our business. Among the issues we are addressing are carbon impact, paper specifications and procurement, ethical conduct within our business and among our vendors, and community and charitable support. For more information, please visit our website: www.wiley.com/go/citizenship.

Copyright © 2011 by John Wiley & Sons, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923 (Web site: www.copyright.com). Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, (201) 748-6011, fax (201) 748-6008, or online at: www.wiley.com/go/permissions.

Evaluation copies are provided to qualified academics and professionals for review purposes only, for use in their courses during the next academic year. These copies are licensed and may not be sold or transferred to a third party. Upon completion of the review period, please return the evaluation copy to Wiley. Return instructions and a free of charge return shipping label are available at: www.wiley.com/go/returnlabel. Outside of the United States, please contact your local representative.

Library of Congress Cataloging in Publication Data:

Doeppner, Thomas W.
 Operating systems in depth / Thomas W. Doeppner.
 p. cm.
 Includes index.
 ISBN 978-0-471-68723-8 (hardback)
 1. Operating systems (Computers) I. Title.
 QA76.76.O63D64 2010
 005.4'3—dc22

2010034669

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To the memory of my father; Thomas W. Doeppner Sr.

This page intentionally left blank

Preface

The goal of this book is to bring together and explain current practice in operating systems. This includes much of what is traditionally covered in operating-system textbooks: concurrency, scheduling, linking and loading, storage management (both real and virtual), file systems, and security. However, I also cover issues that come up every day in operating-systems design and implementation but are not often taught in undergraduate courses. For example, I cover:

- *Deferred work*, which includes deferred and asynchronous procedure calls in Windows, tasklets in Linux, and interrupt threads in Solaris.
- The intricacies of *thread switching*, on both uniprocessor and multiprocessor systems.
- *Modern file systems*, such as *ZFS* and *WAFL*.
- *Distributed file systems*, including *CIFS* and *NFS* version 4.

AUDIENCE

This book is based on material I've developed over the past 30+ years for my operating-systems course at Brown University and is suitable for a one-semester course for advanced undergraduates and beginning graduate students. Students taking the course at Brown have generally had an introductory course on computer architecture and an advanced programming course. The course investigates in depth what's done in current operating systems, and its significant programming projects make students come to grips with major operating-system components and attain an intimate understanding of how they work.

But certainly not all students in an OS course want to go on to concentrate in the area, let alone work in it. A course based on this text must be accessible to these students as well. This issue is handled at Brown by letting students choose one of two projects (discussed later in this preface). The first, relatively straightforward, project involves writing a user-level threads library, a file-system cache manager, and a simple file system. The second, for the truly interested gung-ho students, is to implement a good portion of a simple but fully functional operating system. (Those portions that are not pedagogically useful for students to write are provided to them.) Students completing this latter project get a half course of additional credit (Brown's full courses and half courses are equivalent to four-credit and two-credit courses in schools using the credit system).

TOPIC COVERAGE

Part of the challenge in writing a textbook is not only choosing the topics to cover but also determining the order of presentation. There is the usual conundrum: to appreciate the individual

topics one must first know how they fit together, but to appreciate how they fit together one must first know the individual topics. Compounding this problem is my belief that students must complete a comprehensive programming project in order to understand and appreciate how operating systems work. Thus the material must be presented so as to make such a project possible.

Chapter 1, Introduction, introduces the reader to what an operating system is. It discusses at an abstract level both how they're structured and what they do and provides a brief history. Then, to make things concrete and to introduce the notions of processes and files, I briefly describe early Unix (Sixth Edition, from 1975). This 35-year-old OS is attractive both because of its simplicity and elegance and because, at the level presented here, it's not all that different from either modern Unix systems or modern Windows.

Exploring the concept of a *process* has traditionally been a fundamental part of an operating systems course. However, the meaning of this term has changed over the years. As used in most of today's operating systems, a process is what was originally called a *computation*. Most systems now use the term *thread* to refer to what was originally meant by *process*. Our introduction to the process concept in Chapter 1 uses the contemporary notion of a process; we spend most of Chapter 2, Multithreaded Programming, discussing threads.

It's really important that students doing an operating-systems project understand concurrency—and by “understand” I mean not simply comprehending the concepts but also becoming proficient in multithreaded programming. Chapter 2 is a tutorial on how to write multithreaded programs, both using POSIX threads and Windows threads. Though the material covers such programming from an application-program perspective, both the concepts and the practice apply to programming within an operating system.

In this chapter, I go over the notion of deadlocks. Earlier operating-systems textbooks have spent a great deal of time on deadlocks, describing algorithms for deadlock detection and avoidance. In practice, such algorithms have little or no role in operating systems, so I simply discuss deadlock avoidance.

Chapter 3, Basic Concepts, covers a collection of topics students may have met in previous courses; they aren't all strictly operating-systems material, but they are essential for understanding operating systems. Included here is a discussion of procedure calls and various forms of context switching at an assembler-language level. We talk a lot about context switching and its cost when discussing operating systems, so students really must understand what's involved. This discussion is given both in terms of x86 assembler language and SPARC assembler language. The latter is done to balance the x86-orientation of this part of the chapter but may safely be skipped. Also included in the chapter is a rather simple discussion of I/O architectures, important both for programming projects and for understanding I/O handling in operating systems. I cover dynamic storage-allocation algorithms, following in part Knuth's classic treatment, with additional material on slab allocation. Though this topic could well be considered appropriate for an algorithms course, it doesn't seem to be taught there and it's definitely important in operating systems. I discuss linkers and loaders, concepts that also seem to be much neglected elsewhere in the curriculum. Finally, I go over how a system is booted.

Up to Chapter 4, Operating-System Design, I really haven't explained what's in an operating system, except briefly in the introduction. In Chapter 4, I go over standard operating-system components, discuss flow of control in terms of both threads and interrupt handling, and introduce structuring techniques for operating systems such as monolithic kernels, microkernels, and virtual machines. I talk about some of the layering that's used, such as the virtual file system (VFS) layer in Unix systems and its equivalent in Windows. The material in Chapter 4 is intended to be presented as students begin to design and implement their OS projects; it helps them get started writing the lower-level parts of the operating system.

Concurrency is not merely the use of threads: interrupt handling is a paramount issue, and so are the various forms of deferred processing implemented in many operating systems. I cover these in Chapter 5, Processor Management. This chapter begins by describing the various

approaches to implementing threads packages, from strictly user-level approaches to kernel and hybrid approaches, including scheduler activations. It then covers the design of a simple threads implementation, first on a uniprocessor, then on a multiprocessor. I cover such concepts as preemptive and non-preemptive kernels and how they affect synchronization. This leads naturally to a discussion of scheduling, where I introduce traditional terminology and models (FIFO, SJF, etc.) and then describe approaches to scheduling used in real operating systems. I conclude with descriptions of the schedulers of Linux and Windows. I omit any mention of queuing theory, since in my experience it takes weeks to get to any non-intuitively obvious results, and it's simply not worth the time.

Chapter 6, File Systems, provides extensive coverage of an important part of operating systems. I start, as I started the book, with Sixth-Edition Unix, this time looking at its file system, which came to be called S5FS (the System 5 File System) in later Unix systems. It's amazingly simple, even simplistic, but it's easily understood and even easily implemented as part of a programming project, and it's also a good vehicle for introducing some basic file-system issues. In addition, its performance and stability limitations are inspirations to look at how later file systems have overcome them.

Then, using S5FS as motivation, I cover the basics of on-disk file-system organization, looking at such ideas as cylinder groups, extents, dynamic allocation of on-disk data structures, and log-structured systems. I then cover crash resilience and recovery, focusing on soft updates and journaling. Next comes managing the name space, including a discussion of how directories are organized. This is followed by a discussion of RAID techniques, and then a discussion of the use of flash memory to support file systems. Throughout the file-system discussion, I use examples from a number of file systems, in particular BSD's FFS, Linux's Ext3 and ReiserFS, Microsoft's NTFS, Network Appliance's WAFL, and Sun's ZFS. So that readers can appreciate how all the pieces fit together, I finish with further treatment of some of the example file systems and fill in further details of how they work. An instructor may certainly use her or his judgment in skipping some of the detailed file-system coverage, though I recommend that FFS, NTFS, and ZFS be covered.

Chapter 7, Memory Management, covers both hardware and software aspects of memory management in computer systems. Though many students take computer architecture courses before the OS course, it is still important to discuss the hardware architecture behind memory management—page tables, caches, etc. Thus I devote many pages to this here, looking at a number of representative approaches. It's useful not just to describe the architectures but to explain why they're designed as they are. So I cover the layout of address spaces, showing how support for multiple memory regions with sometimes sparse use drives the design of the memory-management system. I conclude this part of the chapter with a discussion of hardware support for virtualization. Then comes a discussion of the software issues, particularly those having to do with managing virtual memory. This includes not just the usual page-replacement strategies but also backing-store issues, such as space management, as well as handling the various forms of memory-mapped files (both shared and private, in Unix terminology). An important concern is the interplay between memory-mapped file I/O and system-call file I/O.

Chapter 8, Security, covers security as provided by operating systems. It starts with a brief discussion on the threats that modern operating systems try to handle. The bulk of the chapter talks about security architectures, starting with the access-control models developed in the 1960s and '70s. I then discuss the discretionary access-control models used in standard Windows and Unix systems. Next I cover mandatory access control and its use in SELinux. I conclude with a discussion of capability systems.

Chapter 9, Introduction to Networking, provides a rather brief introduction to network protocols (primarily TCP/IP) and remote-procedure protocols. This is clearly a review for students who have taken a networking course. For those students who haven't, it helps form a basis for the following chapter. An instructor pressed for time may safely omit most of the discussion of network protocols, though the RPC material is assumed in the next chapter.

Chapter 10, Distributed File Systems, introduces the issues by looking at the earliest versions of NFS, using it much as S5FS was used in Chapter 6. Here I discuss stateless vs. stateful servers, single-system semantics, failure semantics, and so forth, by taking examples from OSF's DFS (which was based on CMU's AFS), Microsoft's CIFS, and the most recent version of NFS, version 4.

PROGRAMMING PROJECTS

Programming assignments are essential in an OS course. Courses using this text can use either Unix or Windows, though, as described below, we use Linux at Brown. In the past, my students did projects on a simulator that provided an architecture with simple I/O facilities and paged memory management with a software-managed TLB. One year, we had students do projects on Xen-based x86 virtual machines but found that its paravirtualization of I/O deprived students of the experience of working with actual devices. Our current projects are done on top of Bochs, an emulator for the Intel x86 architecture.

A sensitive issue is the choice of programming language. To a first approximation, all current operating systems are written in C. Attempts have been made to write operating systems in object-oriented languages—for example, Sun's Spring OS was written in C++, and in the mid-'90s I based my course's project on Spring—but such approaches never caught on. Though many CS departments (including Brown's) use managed languages, such as Java and C# in the introductory curriculum, these have yet to be used within successful operating systems (though Microsoft Research is doing interesting work on OS development with C#). It's essential that students who go on to work with real operating systems have experience with the language these systems are written in—questions about C constantly come up in job interviews. For all these reasons, all my code examples and exercises use C. For students who don't know C, a quick "introduction to C for Java programmers" at the beginning of the course has proven to be sufficient.

Though the programming projects we do at Brown are not an essential part of the text, we have made our source code and written handouts available at the book's website. As mentioned above, some of our students do more work (for more credit) than others. With that in mind, here is the list of programming assignments.

1. **Shell.** Students write a simple Unix shell in C. This is essentially a warm-up exercise to make sure they are comfortable with C. It also familiarizes them with using basic Unix system calls. What they need to know about Unix is covered in Chapter 1.
2. **Threads.** This is a rather elaborate multithreaded program in which students implement a simple database (containing names and values) as an unbalanced binary search tree. It is accessed and modified concurrently by any number of threads, which can be interrupted by signals and timeouts. The intent is to get students comfortable with concurrent programming (using POSIX threads, though it could be done with Win-32 threads) and with dealing with asynchronous events. The material needed for this is covered in Chapter 2.

Then, students not doing the full OS project do the following:

3. **Threads implementation.** Students implement a simple user-level threads package. They must deal with simple scheduling and synchronization. Much of this material is covered in Chapter 3, with additional topics covered in Chapter 5.
4. **VFS.** We provide a binary of a working OS, minus some features. The first feature they implement is the virtual-file-system layer, in which they provide the high-level (file-system-independent) aspects of doing I/O. This material is covered in Chapters 1 and 4.
5. **S5FS.** They implement the System 5 file system—a simple file system that is based on, if not identical to, the original Unix file system (and not that different from FAT-16 of MS-DOS). This material is covered in Chapter 6.

Students doing the full OS project (known as *Weenix*) do the following:

3. **Kernel 1.** Weenix is implemented bottom-up: students start with device drivers for the terminals and the disk. What they need to know is covered in Chapters 3 and 4.
4. **Kernel 2.** The next step is the implementation of threads and processes. This includes creation and termination, synchronization, and scheduling. This material is covered in Chapter 5.
5. **VFS.** This is identical to the VFS described above and is covered in Chapters 1 and 4.
6. **S5FS.** This is identical to the S5FS described above and is covered in Chapter 6.
7. **Virtual memory and final integration.** Students implement a simple paging system (no page-outs, just page-ins). They then get all the components to work with each other so as to support multiple user-level processes, which run the shell they implemented in the first assignment. This material is covered in Chapter 7.

We tried a few years ago to create a version of the OS project that was based on a simplification of Windows NT rather than on a simplification of Unix. Though we had what we thought was a rather elegant design, it proved to be a much larger system in terms of code size than the Unix-based system. Unix started as a small system, while Windows NT did not. Thus it is not surprising that a usable subset of Unix is much smaller than a usable subset of Windows. Our Windows subset was just large enough that it was unsuitable for a one-semester project. Therefore we continue with our Unix-based project.

ADDITIONAL MATERIALS

Much additional material is available at the book's web site: www.wiley.com/college/Doeppner. Available to students are programming exercises, with source code, for Chapter 2 (Multithreaded Programming). Available to faculty are PowerPoint lecture slides (not just the figures from the text but the slides I use to teach my operating-systems course), answers to all exercises, and full source code, handouts, and other explanatory material for the optional semester-long operating-systems project.

EXERCISES

The end-of-chapter exercises are of three types: unstarred, starred (*), and two-starred (**). Unstarred ones are intended to be easy and are provided for quick review of the material. Starred and two-starred exercises are intended to be homework problems and exam questions. The two-starred ones are both more difficult and more valuable than the one-starred ones.

ACKNOWLEDGMENTS

A lot of people have helped me prepare this book. I particularly want to thank all the students who have been teaching assistants in my operating-systems course over the years. The following former students, all of whom were also TAs, have continued to help out, giving me useful information on the products of their employers (of which they were often principal architects): Matt Ahrens, George Cabrera, Bryan Cantrill, Kit Colbert, Peter Griess, Adam Leventhal, Dave Pacheco, and Mike Shapiro. Aaron Myers and Eric Tamura provided considerable help by writing many of the exercises. The following former students provided useful criticisms and bug reports on the text while in my course: Lucia Ballard, Sam Cates, Adam Conrad, Andrés Douglas, Colin Gordon, Alex Heitzmann, Daniel Heller, Venkatasubramanian Jayaraman, Daniel Leventhal, Chu-Chi Liu, Tim O'Donnell, Daniel Rosenberg, Allan Shortlidge, Sean Smith, Jono Spiro, Taylor Stearns, and Zhe Zhang.

The following Brown students and alums have put in significant time and effort creating and fixing up the project I use in my course at Brown, which is available on the book's web site: Keith Adams, Dimo Bounov, Michael Castelle, Adam Fenn, Alvin Kerber, Dan Kuebrich, Jason Lango, Robert Manchester, Dave Pacheco, David Powell, Eric Schrock, Chris Siden, Shaun Verch, and Joel Weinberger.

I greatly appreciate the following Microsoft employees who, with much patience, answered my frequent questions about Windows: Mark Eden, Susheel Gopalan, Dave Probert, Ravisankar Pudipeddi, Arkady Retik, Scott Williams, and Mark Zbikowski.

Bob Munck, a Brown alumnus from before I came to Brown, helped me track down historical information about virtual machines. The following Brown faculty and staff all assisted me in many ways: John Bazik, Steven Carmody, Mark Dieterich, Maurice Herlihy, John Hughes, John Jannotti, Philip Klein, Steve Reiss, Max Salvas, and Andy van Dam.

Rosemary Simpson did an amazing job of constructing the index, the URL index, and the glossary. She also checked and corrected many of my references. Any errors that remain are certainly my fault and not hers.

This book took a long time from conception to completion. During that period, I had a number of editors at Wiley, all of whom I wish to thank: Paul Crockett, Beth Golub, Dan Sayre, Bruce Spatz, and Bill Zobrist. It has been a pleasure working with Barbara Russiello, who guided the book through production at Wiley.

I wish to thank the following reviewers for their helpful comments: Bina Ramamurthy, Hugh C. Lauer, Kenneth F. Wong, Kyoungwon Suh, Tim Lin, and Euripides Montagne.

I am grateful to Microsoft for providing funding as I began the book project.

Much of the material on multithreaded programming in Chapter 2 was developed as part of training courses I have produced and taught for the Institute for Advanced Professional Studies in Boston, Massachusetts. I thank Donald French, the company's president, for his support in this.

Most of all, I couldn't have done it without my copy editor, Katrina Avery, who is also my wife.

Contents

1	Introduction	1
1.1	Operating Systems	2
1.1.1	Operating Systems as a Field of Study	3
1.2	A Brief History of Operating Systems	4
1.2.1	The 1950s: The Birth of the Concept	4
1.2.2	The 1960s: The Modern OS Takes Form	5
1.2.3	Minicomputers and Unix	6
1.2.4	The Personal Computer	8
1.3	A Simple OS	12
1.3.1	OS Structure	12
1.3.2	Processes, Address Spaces, and Threads	14
1.3.3	Managing Processes	16
1.3.4	Loading Programs into Processes	19
1.3.5	Files	20
1.4	Beyond a Simple OS	33
1.4.1	Extensions	34
1.4.2	New Functionality	35
1.5	Conclusions	37
1.6	Exercises	37
1.7	References	38
2	Multithreaded Programming	39
2.1	Why Threads?	40
2.2	Programming with Threads	44
2.2.1	Thread Creation and Termination	44
2.2.2	Threads and C++	53
2.2.3	Synchronization	56
2.2.4	Thread Safety	76
2.2.5	Deviations	79
2.3	Conclusions	88
2.4	Exercises	88
2.5	References	92
3	Basic Concepts	93
3.1	Context Switching	93
3.1.1	Procedures	94

3.1.2	Threads and Coroutines	100
3.1.3	System Calls	102
3.1.4	Interrupts	102
3.2	Input/Output Architectures	104
3.3	Dynamic Storage Allocation	107
3.3.1	Best-Fit and First-Fit Algorithms	107
3.3.2	Buddy System	108
3.3.3	Slab Allocation	109
3.4	Linking and Loading	109
3.4.1	Static Linking and Loading	109
3.4.2	Shared Libraries	116
3.5	Bootting	119
3.6	Conclusions	123
3.7	Exercises	123
3.8	References	125
4	Operating-System Design	126
4.1	A Simple System	127
4.1.1	A Framework for Devices	129
4.1.2	Low-Level Kernel	131
4.1.3	Processes and Threads	137
4.1.4	Storage Management	139
4.2	Rethinking Operating-System Structure	143
4.2.1	Virtual Machines	145
4.2.2	Microkernels	155
4.3	Conclusions	160
4.4	Exercises	161
4.5	References	162
5	Processor Management	163
5.1	Threads Implementations	164
5.1.1	Strategies	164
5.1.2	A Simple Threads Implementation	170
5.1.3	Multiple Processors	172
5.2	Interrupts	177
5.2.1	Interrupt Handlers	178
5.2.2	Deferred Work	183
5.2.3	Directed Processing	187
5.3	Scheduling	192
5.3.1	Strategy	193
5.3.2	Tactics	204
5.3.3	Case Studies	206
5.4	Conclusions	212
5.5	Exercises	212
5.6	References	215
6	File Systems	217
6.1	The Basics of File Systems	218
6.1.1	Unix's S5FS	218
6.1.2	Disk Architecture	223
6.1.3	Problems with S5FS	226

6.1.4 Improving Performance	227
6.1.5 Dynamic Inodes	236
6.2 Crash Resiliency	237
6.2.1 What Goes Wrong	238
6.2.2 Dealing with Crashes	240
6.3 Directories and Naming	253
6.3.1 Directories	254
6.3.2 Name-Space Management	261
6.4 Multiple Disks	263
6.4.1 Redundant Arrays of Inexpensive Disks (RAID)	266
6.5 Flash Memory	269
6.5.1 Flash Technology	269
6.5.2 Flash-Aware File Systems	270
6.5.3 Augmenting Disk Storage	271
6.6 Case Studies	271
6.6.1 FFS	271
6.6.2 Ext3	272
6.6.3 Reiser FS	273
6.6.4 NTFS	275
6.6.5 WAFL	276
6.6.6 ZFS	278
6.7 Conclusions	282
6.8 Exercises	283
6.9 References	285
7 Memory Management	287
7.1 Memory Management in the Early Days	287
7.2 Hardware Support for Virtual Memory	289
7.2.1 Forward-Mapped Page Tables	291
7.2.2 Linear Page Tables	292
7.2.3 Hashed Page Tables	294
7.2.4 Translation Lookaside Buffers	296
7.2.5 64-Bit Issues	297
7.2.6 Virtualization	299
7.3 Operating-System Issues	301
7.3.1 General Concerns	301
7.3.2 Representative Systems	305
7.3.3 Copy on Write and Fork	310
7.3.4 Backing Store Issues	313
7.4 Conclusions	315
7.5 Exercises	315
7.6 References	318
8 Security	319
8.1 Security Goals	319
8.1.1 Threats	320
8.2 Security Architectures	323
8.2.1 Access Control in Traditional Systems	325
8.2.2 Mandatory Access Control	334
8.2.3 Capability Systems	342

8.3	Conclusions	346
8.4	Exercises	347
8.5	References	348
9	Introduction to Networking	350
9.1	Network Basics	350
9.1.1	Network Protocols	351
9.2	Remote Procedure Call Protocols	364
9.2.1	Marshalling	366
9.2.2	Reliable Semantics	368
9.3	Conclusions	374
9.4	Exercises	374
9.5	References	377
10	Distributed File Systems	378
10.1	The Basics	380
10.2	NFS Version 2	382
10.2.1	RPC Semantics	386
10.2.2	Mount Protocol	386
10.2.3	NFS File Protocol	387
10.2.4	Network Lock Manager	388
10.3	Common Internet File System (CIFS)	389
10.3.1	Server Message Block (SMB) Protocol	391
10.3.2	Opportunistic Locks	392
10.4	DFS	394
10.5	NFS Version 4	397
10.5.1	Managing State	398
10.5.2	Dealing with Failure	401
10.6	Conclusions	402
10.7	Exercises	403
10.8	References	404
	Appendix Index of URLs	405
	Index	407

1.1	Operating Systems	1.3.5	Files
1.1.1	Operating Systems as a Field of Study	1.3.5.1	Naming Files
1.2	A Brief History of Operating Systems	1.3.5.2	Using File Descriptors
1.2.1	The 1950s: The Birth of the Concept	1.3.5.3	Random Access
1.2.2	The 1960s: The Modern OS Takes Form	1.3.5.4	Pipes
1.2.3	Minicomputers and Unix	1.3.5.5	Directories
1.2.4	The Personal Computer	1.3.5.6	Access Protection
1.2.4.1	Hobbyist Computing	1.3.5.7	Creating Files
1.2.4.2	Computer Workstations	1.4	Beyond a Simple OS
1.2.4.3	Microsoft Windows	1.4.1	Extensions
1.2.4.4	Unix Continues	1.4.1.1	Multithreaded Processes
1.3	A Simple OS	1.4.1.2	Virtual Memory
1.3.1	OS Structure	1.4.1.3	Naming
1.3.1.1	Traps	1.4.1.4	Object References
1.3.1.2	System Calls	1.4.1.5	Security
1.3.1.3	Interrupts	1.4.2	New Functionality
1.3.2	Processes, Address Spaces, and Threads	1.4.2.1	Networking
1.3.3	Managing Processes	1.4.2.2	Interactive User Interfaces
1.3.4	Loading Programs into Processes	1.4.2.3	Software Complexity
		1.5	Conclusions
		1.6	Exercises
		1.7	References

We begin by explaining what an operating system is and why the field is not only worthy of study, but worthy of continued research and development. Next we delve briefly into the history of operating systems, one that combines brilliant research in both academia and industry with a colorful cast of characters. Finally we introduce the study of operating systems by looking carefully at an early version of Unix so as to understand its basic concepts and some of how it was implemented.

1.1 OPERATING SYSTEMS

What's an operating system? You might say it's what's between you and the hardware, but that would cover pretty much all software. So let's say it's the software that sits between your software and the hardware. But does that mean that the library you picked up from some web site is part of the operating system? We probably want our operating-system definition to be a bit less inclusive. So, let's say that it's that software that almost everything else depends upon. This is still vague, but then the term is used in a rather nebulous manner throughout the industry.

Perhaps we can do better by describing what an operating system is actually supposed to do. From a programmer's point of view, operating systems provide useful abstractions of the underlying hardware facilities. Since many programs can use these facilities at once, the operating system is also responsible for managing how these facilities are shared.

To be more specific, typical hardware facilities for which the operating system provides abstractions include

- processors
- RAM (random-access memory, sometimes known as *primary storage*, *primary memory*, or *physical memory*)
- disks (a particular kind of *secondary storage*)
- network interface
- display
- keyboard
- mouse

We like to think of a program as having sole use of a processor and some memory, but in reality it's sharing the processor with other programs. In addition, what the program perceives as "memory" may be partly on actual RAM and partly on disk. Rather than force the programmer to worry about all this, most operating systems provide the *process* abstraction, which captures the notions of both an execution unit (a processor) and memory.

Similarly, rather than making programmers worry about the myriad arcane details of dealing with disk storage and buffering its contents in RAM, some sort of simple *file* abstraction is provided for persistent storage.

Network interfaces provide access to computer networks such as Ethernet. The sorts of *networking* abstractions used by application programmers include *sockets*, *remote procedure calls*, and *remote method invocation*, as well as *web-oriented interactions*. Later we discuss in detail the multiple layers of abstraction used with networks.

The display, keyboard, and mouse require layers of software and hardware in order to have useful abstractions. Few programmers want to think of the display as an output device to which they send bit patterns to turn pixels on and off selectively. Instead, we use abstractions provided by drawing packages, windowing packages, and the like. A mouse provides a steady stream of position information and button-click indications. A keyboard provides indications that keys are being depressed and released. Abstractions are supplied for both devices so that, for example, we can treat mouse clicks as invocations of programs and key-down events as inputs of characters.

Some of these abstractions, usually because of their history, are considered part of fields other than operating systems. For example, file systems are generally considered part of the operating system, but databases form a field unto themselves. And though the low-level aspects of

displays are considered as abstractions provided by operating systems, the higher-level graphics and windowing systems are again aspects of fields unto themselves.

Abstracting (or “virtualizing”) the various components of our computing systems also makes it relatively straightforward to share them. Rather than having to worry about sharing both the sectors of a disk drive and the bandwidth through its controller, we simply deal with files. Programs create and name files, add data to them, read the data back, delete them: there’s no explicit concern about actually sharing the physical disk resources (until, of course, we run out of disk space).

In addition, by using the *process* abstraction, which abstracts both processors and memory, programmers need only be concerned about the program at hand. They needn’t worry about whether this program is monopolizing the processor or is using all of RAM: both the processor and RAM are multiplexed under the control of the operating system.

1.1.1 OPERATING SYSTEMS AS A FIELD OF STUDY

Why should we study operating systems? The field has been a standard part of the computer science curriculum ever since there was a computer science curriculum. One might figure that it is a solved problem and it is time to move on.

To a certain extent, it is a solved problem. We know how to build an operating system that supports a number of concurrently executing programs that, depending on our intent, can be completely oblivious of one another’s existence, fully cooperating with one another, or some combination of the two. We can interact with programs running on a computer as if the entire computer were dedicated to us, even though much else is going on. Bugs in programs rarely bring down the entire system. File systems are amazingly efficient, both in time and in space. Our systems and data can survive hardware failures. Vast amounts of data can be transferred to and from our computer over networks flawlessly.

Yet things are not always so perfect, particularly when our systems are pushed to their limits. Systems are routinely attacked and succumb to their attackers. Operating systems occasionally crash due to bugs in application programs. A system might be so overloaded, perhaps due to a “denial-of-service attack,” that it is effectively down. Data does occasionally get lost or corrupted.

The challenge in the field of operating systems is to continue to strive towards overcoming these problems. We like to think that our basic approaches to system design are sound and all that is required are better implementations and minor tweaking, but it may well be that completely new approaches are required. A common theme in current operating-system development is isolation: programs should be isolated from one another so that problems in one do not affect others. This has been a theme since the earliest days of operating-system development and yet continues to be a solved problem that is never solved quite well enough.

In the past isolation meant protection from bugs. A bug in one computation should not bring down another. Data was to be kept reasonably secure in file systems, but real security meant locking the file systems in vaults with no outside network connections. Today’s operating systems must cope with attacks from national intelligence services, organizations that might well understand a system’s limitations, bugs, and security holes far better than its designers do.

We seem to be moving towards a global computing environment in which data is stored in “the cloud.” Yet this cloud resides on file systems that are not unlike those that exist on your laptop computer, managed by an operating system that might be identical to that on your laptop. Operating systems continue to be relevant, even if much of the action is moving away from personal computers.

1.2 A BRIEF HISTORY OF OPERATING SYSTEMS

In this section we present a brief history of operating systems. We start with the early days, when the notion of an operating system had yet to be invented, and proceed to the systems prevalent today, concentrating on representative systems with clear relevance to today's systems.

1.2.1 THE 1950s: THE BIRTH OF THE CONCEPT

The earliest computers had no operating systems. In fact, they had very little software, period. When you ran a program on an early computer, you supplied everything — there weren't even any program libraries.

The early history of operating systems is a bit murky. The term “operating system” was probably coined sometime after the first few software systems that might be called operating systems came into existence. According to MIT's CSAIL web pages,¹ MIT's first operating system, apparently for its Whirlwind computer, was written in 1954 and was used to manage the reading of paper tapes so that the computer could run without a human operator being present.

General Motors Research Laboratories also claims to have written the first operating system, in partnership with North American Aviation, for the IBM 701 and then the IBM 704. According to (Ryckman 1983), this operating system ran on the IBM 704 in May 1956 and was simply called “Input/Output System.” Its motivation was that users of the IBM 701 were allocated time in 15-minute slots, but ten minutes was needed to set up the necessary equipment, such as tapes, card readers, etc. The goal of the system was to make this setup work unnecessary. Though it may have been the precursor of the modern operating system, it apparently was not considered an important development at the time — it gets scant mention (five short lines) in just one article in a special issue of *Annals of the History of Computing* (Ryckman 1983) devoted to the IBM 701.

The earliest serious paper discussing operating systems may be the one presented at a conference in 1959 by Christopher Strachey, later of Oxford (Strachey 1959).² Though the title of the paper was “Time Sharing in Large Fast Computers,” the subject actually was what we now call multiprogramming, and not interactive computing. ((Lee 1992) discusses this early confusion of terms.) In a January 1959 memo,³ John McCarthy, then of MIT, proposed that MIT “time-share our expected ‘transistorized IBM 709’” (referring to the IBM 7090) (McCarthy 1983). Here, “time-share” was used in its modern sense, meaning interactive computing by multiple concurrent users.

The earliest operating systems were batch systems, designed to facilitate the running of multiple jobs sequentially. A major bottleneck in the earliest computers was I/O: all computation had to stop to let an I/O operation take place. What was necessary to eliminate this problem was to enable computers to start an operation, but not wait for it to complete and, correspondingly, to find out when an operation completed. Aiding the latter is the notion of the *interrupt*. According to (Smotherman 2008), the first machine to implement interrupts was the DYSEAC, designed and built by the U.S. National Bureau of Standards in 1954. The concept was added to the Univac 1103 and in 1955 became a standard feature of the 1103a. Edsger Dijkstra, whom we encounter again in Chapter 2, designed a system with interrupts as part of his 1959 Ph.D. thesis (Dijkstra 1959). According to (Smotherman 2008), this work involved the Electrologica X-1 computer and was done in 1957–1958.

¹ <http://www.csail.mit.edu/timeline/timeline.php?query=event&id=3>

² There is some confusion about the date: Strachey himself referred to it later as being presented in 1960, but it really seems to have been presented in 1959. I have been unable to find a copy of the paper and must rely on what others have written about it.

³ McCarthy is not entirely sure about the date; he states that it might have been in 1960.

The ability to compute and perform I/O concurrently made possible what we now call *multiprogramming* — the concurrent execution of multiple programs. Or, perhaps more precisely, multiprogramming made it possible for a system to compute and perform I/O concurrently, since while one program was doing I/O, another could be computing. While the second-generation operating systems were still batch systems, they supported multiprogramming.

Information about the actual operating systems that supported early multiprogramming is sparse, but early systems with such support include the Lincoln Labs TX-2 (1957) and the IBM Stretch computer, also known as the IBM 7030, developed in the latter half of the 1950s and first delivered in 1961. The Stretch was more expensive and slower than planned, but was nevertheless the world's fastest computer from 1961 till 1964 (Wikipedia 2009).

1.2.2 THE 1960s: THE MODERN OS TAKES FORM

While operating systems were essentially an afterthought in the 1950s, they drove computer development throughout the 1960s, perhaps the most interesting decade of operating-system development, beginning with the first systems to support virtual memory and ending with the earliest Unix system. In between came OS/360 and Multics.

As the decade began, memory devices were commonly thought of as in a hierarchy, from the fastest and most expensive — primary storage — to slower but cheaper — secondary storage — to archival storage (even slower and even cheaper). Primary storage was typically built on core memory,⁴ while secondary storage was built on disks and drums.⁵

It was often not possible to fit a program in primary storage, and thus code and data had to be shuffled between primary storage and secondary storage. Doing this shuffling explicitly within a program was, at best, a nuisance. Researchers at the University of Manchester (in the United Kingdom) incorporated in the Atlas computer a *one-level store* (Kilburn, Edwards, et al. 1962): programmers wrote programs as if lots of primary storage was available, while the operating system shuffled code and data between the relatively small quantity of actual primary storage and the larger amount of secondary storage. This notion became known as *virtual memory*, a subject we take up in Chapter 7.

At roughly the same time, researchers at MIT pursued the notion of time-sharing: enabling multiple concurrent users to interact with programs running on a computer. They developed CTSS (Corbató, Daggett, et al. 1962), their compatible time-sharing system — compatible in being compatible with a standard batch system for the IBM 709/7090 computer — FMS (Fortran Monitor System). It supported three concurrent users as well as an FMS batch stream. Though CTSS was installed only at MIT, it helped prove that time-sharing made sense.

Perhaps the most influential system to come out of MIT was Multics (multiplexed information and computing service). Multics started in 1964 as a joint project of MIT, General Electric, and Bell Labs; Bell Labs dropped out in 1969 and GE's computer business was purchased in 1970 by Honeywell, who took over the commercialization of Multics. The goals of the Multics project, as stated in (Corbató, Saltzer, et al. 1972), were to build a computer utility with

1. convenient remote terminal access as the normal mode of system usage;
2. a view of continuous operation analogous to that of the electric power and telephone companies;

⁴“Core memory” is memory formed from magnetic toruses, called cores. Each held one bit and was the dominant technology for primary storage from the late 1950s through the early 1970s. Primary memory is still often referred to as core, as in “core dump.”

⁵A drum is similar to a disk. If a disk resembles a stack of old-fashioned phonograph records, a drum resembles Edison's even older phonograph cylinder. However, drums had one head per track and could access data more quickly than disks.

3. a wide range of capacity to allow growth or contraction without either system or user reorganization;
4. an internal file system so reliable that users trust their only copy of programs and data to be stored in it;
5. sufficient control of access to allow selective sharing of information;
6. the ability to structure hierarchically both the logical storage of information as well as the administration of the system;
7. the capability of serving large and small users without inefficiency to either;
8. the ability to support different programming environments and human interfaces within a single system;
9. the flexibility and generality of system organization required for evolution through successive waves of technological improvements and the inevitable growth of user expectations.

This would be an impressive set of goals today; in 1965 it was amazing. Multics also was one of the first, if not the first, operating system to be written in a high-level language, PL/1. Operating systems had always been written in assembler language, mainly because it was thought (probably correctly) that compilers did not produce code efficient enough for an operating system.

Multics employed a one-level store, unifying the notions of file and virtual memory so that files were simply objects that were mapped into memory. Its access control system was advanced for its day; much early work in computer security was done in the context of Multics systems.

Even though Multics was highly influential and continued to be used until 2000, it was not a commercial success. Because of its importance in the secure-computing community, it was used in many environments requiring access to U.S. government secrets, but it found little application elsewhere, perhaps because it required special hardware features unavailable on other computers.

The other important operating system of the era was OS/360, which actually was commercially successful. It was designed on the premise that one operating system should run on a family of machines, from small machines of little capacity to large machines of enormous capacity. Thus applications running on it would be portable across the family of machines. The machine family was, of course, the IBM 360, which did have a wide range of performance.

Unfortunately, the project did not turn out as planned. Rather than one operating system, a family of similar operating systems was created with functionality dependent on the performance of the targeted machines.

What the OS/360 development project made painfully evident was that producing such an operating system required an enormous effort by a large group of people. Many of the lessons learned from the project were described in the 1975 book *The Mythical Man-Month* by the project's leader, Fred Brooks. The most famous of these lessons, inspiring the book's title, was that a task requiring twelve months of one person's time just could not be done in one month by twelve people.

1.2.3 MINI COMPUTERS AND UNIX

IBM, particularly in the 1960s and 1970s, was associated with *mainframe* computers — physically large computers that served entire organizations. It had a number of competitors in this market, but most fell by the wayside. In contrast to such mainframes were *minicomputers* — physically large by today's standards but puny compared to mainframes. The first, the PDP-8 from the Digital Equipment Corporation (DEC), appeared in the mid-1960s, was small and sufficiently inexpensive that it could be used by small laboratories (including those of fledgling

academic computer science departments). Minicomputers became increasingly popular in the late 1960s and throughout the 1970s, with the introduction of the Nova from Data General and the PDP-11 from DEC. These, and others from a number of companies, proved useful for more general-purpose computing. A fair number of operating systems were written for them — in some cases a number of different operating systems for the same computer — making the 1970s the heyday of operating-system development.

The most famous of these minicomputer operating systems, and probably the only one still extant in any form, is Unix. Its early history is legendary. Among the people at Bell Laboratories participating in the Multics project until Bell Labs dropped out were Ken Thompson and Dennis Ritchie. In 1969 Thompson designed and implemented a simple operating system on a spare DEC PDP-7 (apparently stored in a closet). He was joined by Dennis Ritchie, and they gave the system the name Unix — punning on Multics and indicating that it was nowhere near as grandiose as Multics. It was later re-implemented on the much more capable PDP-11. A community of researchers at Bell Labs joined in, adding both systems and applications software.

An important aspect of Unix was that, except for its earliest implementations, it has been written in C. The history of C is interesting in itself. It traces its history to CPL (Combined Programming Language — “combined” because it combined efforts from Cambridge University and the University of London), among whose authors was the aforementioned Christopher Strachey. CPL begat BCPL (Basic CPL), a language intended for systems programming. BCPL begat B, a version of BCPL that was sufficiently stripped down (at Bell Labs) that its compiler could run on a minicomputer; B was used to implement some early versions of Unix. B begat C (second letter of BCPL), a version of B that was expanded so as to implement (also at Bell Labs) an early computer game (“Space Travel”). The de facto standard for C became the book *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, first published in 1978, and known simply as “K&R.”

Unix evolved within Bell Labs, each new version being released as a new edition. Sixth-Edition Unix was made available to universities in 1975 very inexpensively: a few hundred dollars bought the complete source code of the system, with the restriction that the source code must be treated essentially as a trade secret and that the system could not be used for administrative purposes. These were very attractive terms and a number of university computer science departments took advantage of them. Later, in 1978, Seventh-Edition Unix was released on similar terms.

At around this time, minicomputer manufacturers came out with “superminis” — even more capable minicomputers that supported virtual memory. Of particular interest was DEC’s VAX-11/780, with a compatibility mode in which it could run PDP-11 code. A group at Bell Labs ported Seventh-Edition Unix to it to create a product known as Unix 32V, which was released to the academic and research communities on the same terms as Seventh-Edition Unix.

However, Unix 32V did not exploit a number of the features of the VAX-11/780, particularly virtual memory: it simply treated the VAX as a faster PDP-11. A group at the University of California, Berkeley, received an early copy of Unix 32V and from it produced a version that did take advantage of virtual memory — this was released to the academic and research communities as 3 BSD (third Berkeley Software Distribution).⁶ 3 BSD did not perform very well, but the effort at Berkeley continued with the fourth Berkeley Software Distribution, which made a number of performance improvements — a major goal was for it to perform as well as VMS, DEC’s operating system for the VAX-11 series. (VMS was itself an important operating system and was just as advanced for its day as Unix.) Further improvements came with 4.1 BSD. By this time the Berkeley group had received funding from DARPA (Defense Advanced Research Projects Agency, an important agency within the U.S. Department of Defense that funded much

⁶What were the first and second Berkeley software distributions? They were packages of application programs made available to the Unix community for PDP-11s.

computer science research) to improve the operating system so it could support research on the nascent ARPAnet (the predecessor of the Internet).

The DARPA-supported effort added much functionality to Unix, perhaps the most important of which was support for networking. This was made available through a series of releases: 4.1a, 4.1b, and 4.1c, culminating in 4.2 BSD. The project continued, producing a number of incremental improvements and ultimately 4.4 BSD (McKusick, Bostic, et al. 1996).

While the Berkeley effort was taking place, other groups within Bell Laboratories continued their own Unix development. These efforts resulted in a version that was offered as a commercial product, Unix System III, followed as a product by Unix System V (apparently System IV didn't make the cut). (Bach 1986), the first book to cover Unix internals, describes the design of early Unix System V. It was decided that Unix System V was a brand with some loyalty, and thus instead of Unix System VI, there was Unix System V Release 2, continuing through Release 4 in 1988. System V Release 4 was actually a major new release, combining aspects of 4.3 BSD and previous versions of Unix System V, as well as Xenix and SunOS.

1.2.4 THE PERSONAL COMPUTER

While the Unix saga was going on, the notion of a personal computer started to take shape. Some of the most important early work, at Xerox's Palo Alto Research Center (Xerox PARC) in the 1970s, essentially invented the whole notion of personal computing, including window-managed displays. Xerox was never able to turn the Xerox PARC work into a commercial success, but it was highly influential and set the stage for the 1980s. Below we first discuss hobbyist computing, then computer workstations. The two had essentially merged by 2000 or so.

1.2.4.1 Hobbyist Computing

The 1970s were the era of hobbyist computing. The operating systems were simple, but they had a lasting influence. One of the first and perhaps the most significant was CP/M (Kildall 1981), completed in 1974 and designed and implemented by Gary Kildall, who went on to found Digital Research to market and develop it further. Originally designed for the Intel 8080, an 8-bit processor, but eventually ported to a number of different architectures, CP/M was primitive, but so was the hardware it ran on. It handled just one application at a time and had no way to protect itself from the application. It did have a simple file system. Among its innovations (in a later release) was the clear separation of architecture-dependent code from architecture-independent code, which facilitated its adaptation to a large number of platforms. CP/M was originally structured as three components: console command processor (CCP), basic disk operating system (BDOS), and basic input/output system (BIOS). Its CCP was the component that users dealt with directly, leading to the common view that the user interface is the operating system.

Somewhat later than CP/M, but just as primitive, was Apple DOS, designed for the Apple II. The original Apple II had no disk. Later versions were given floppies, but no operating system — programs on disk were self-booting and contained their own (very primitive) OSes. Apple DOS 3.1, the first version released to customers, became available in 1978 and had similar functionality to CP/M at the time (i.e., not much). Apple introduced the Apple III computer in 1980 with a somewhat more sophisticated operating system — Apple SOS (sophisticated operating system, pronounced “apple sauce”). The system (hardware plus software) was a commercial failure, though it may have had some influence on later Apple systems.

Microsoft started as a programming-language company; its first product was a Basic interpreter, sold to MITS⁷ for use on the Altair 8800 in 1975. In 1979 the company found it expedient

⁷Micro Instrumentation and Telemetry Systems — a small company founded by the late H. Edward Roberts with no relation to the Massachusetts Institute of Technology.

to buy its way into the operating-system business, and did so by purchasing a license for ... Unix. It acquired a license for Seventh-Edition Unix from AT&T, adapted it for 16-bit microcomputers, and called its version Xenix (AT&T was extremely protective of the Unix name). It was later ported to the Intel 8086 (and still later to the 80286 and the 386) by the Santa Cruz Operation. Xenix was used internally by Microsoft throughout the 1980s and was the prevalent version of Unix during most of this period.

Microsoft is, of course, no longer known as a purveyor of Unix systems; it is more famous for another operating system for which it purchased rights. Microsoft was approached by IBM in 1980 to provide its Basic system for the forthcoming IBM PC. IBM apparently wanted to license CP/M as its operating system, but was unable to agree on terms with Digital Research. However, another company, Seattle Computer Products (SCP), had somewhat earlier produced an operating system for the 8086, QDOS (for “quick and dirty operating system”), that was similar to CP/M.⁸ Microsoft, having no unencumbered operating system of its own (it had to pay royalties to AT&T for Xenix), offered to supply an operating system for the IBM PC and purchased QDOS from SCP when IBM said yes. This became known as PC-DOS when sold by IBM on their computers, and MS-DOS when licensed by Microsoft to other manufacturers for use on their computers.

CP/M, later renamed DR-DOS, was eventually offered by Digital Research as an operating system for IBM PCs and their clones, but never achieved the market share of MS-DOS.

Among the features supported by minicomputer operating systems in the 1980s, but not by operating systems for low-end personal computers, were virtual memory, multitasking, and access control for file systems. *Multitasking* is similar to multiprogramming, but emphasizes the concurrent execution of multiple programs associated with the same user. Some systems supported *cooperative multitasking*, in which the individual programs explicitly pass control to one another. Almost completely absent was any notion of *preemptive multitasking*, in which the operating system automatically preempts the execution of programs to let others execute, thus forcing all programs to share a processor. (We cover this in detail in Chapter 5.)

Access control for file systems is important when a number of people can access them. But, particularly in the early days, personal computers had floppy disks or diskettes, not hard disks. Access control was simple — people were careful with whom they shared floppies. Even with hard disks, people still considered the disk their personal property and saw no need to protect files individually. The notion that an errant program might do damage did not seem to be a concern, and the notion that a virus or other sort of malware might damage unprotected files was unheard of.

Despite its commercial success and continued development by Microsoft, MS-DOS never became as sophisticated as even the early Unix systems. It did not support any form of multitasking (though, as discussed below, the addition of early Windows provided cooperative multitasking), did not protect the operating system from applications, and did not have any form of access control in its file system.

Apple, in the meantime, began development of two new operating systems for its Lisa and Macintosh computers. The Lisa OS was relatively advanced for its time and price range — it supported multitasking, virtual memory, and a hierarchical file system, and had a very advanced graphical user interface influenced by the work at Xerox PARC. But it was completely eclipsed by the cheaper, better advertised Macintosh, which had a much less capable operating system — no multitasking, no virtual memory, and no hierarchical file system. Soon after its initial release in 1984 the Macintosh did get a hierarchical file system, but the other features did not materialize until a completely new operating system was introduced in 2000.

⁸ Just how similar it was has been disputed both in and out of court.

1.2.4.2 Computer Workstations

While the 1980s are known for the birth of the personal computer, both IBM and Apple, also born in the decade was the computer workstation. This was essentially a personal computer as well, but much more expensive and coming more from the minicomputer tradition than the hobbyist tradition. Its operating systems were not just afterthoughts, but state of the art.

Two of the more influential systems were those of Apollo and Sun. Apollo was founded in 1980 and shipped its first workstations in 1982. Their operating system, Aegis, supported multitasking, virtual memory, and the first commercially successful distributed file system, complete with access control. It had a number of advanced features, many of them derived from Multics, and supported bit-mapped graphics and a window manager. Unfortunately for Apollo, their market began to demand Unix, so Apollo was forced to provide Unix emulation to run Unix applications. Apollo was ultimately acquired by Hewlett Packard in 1989, who integrated much of Apollo's technology into their own products.

Sun was founded in 1982 and shipped its first workstations the same year. Its operating system, SunOS, was derived from 4.2 BSD (not surprisingly, since one of its founders, Bill Joy, essentially led the 4.2 BSD development effort while a graduate student at UC Berkeley). In 1984 it introduced NFS (network file system), a distributed file system still in active use (and under active development) today. SunOS was renamed Solaris in 1992, when it was revamped to support Unix System V Release 4 as part of Sun's joint work with AT&T. (Operating-system version numbers do not necessarily follow a logical pattern. Solaris was introduced as the operating system following SunOS 4 and was called Solaris 2.0; the name Solaris 1.0 was applied retrospectively to all the previous releases of SunOS. Solaris releases followed a logical pattern for a while: Solaris 2.1, 2.2, 2.3, etc., but the version following Solaris 2.6 was called Solaris 7, apparently because 2.7 wasn't an impressive enough number.)

1.2.4.3 Microsoft Windows

Microsoft Windows started as an application running on MS-DOS. Windows 1.0, released in 1985, was not an operating system at all, but merely provided support for windows on the display. It did allow cooperative multitasking, in which a number of applications run concurrently and explicitly yield the processor to one another. Windows 2.0 provided some improvements; Windows 3.0 and 3.1 used hardware segmentation to allow programs to access more memory than in earlier versions, but it was still quite possible for one application to overwrite another, and for applications to overwrite the operating system.

Beginning with Windows 95 (released, of course, in 1995), Windows supported preemptive multitasking: concurrently executing programs could share the processor via time-slicing. It also supported something close to virtual memory, allowing programs to run in separate address spaces. However, an errant program could still crash the entire system by writing into operating-system locations. MS-DOS was still present, but much OS functionality had been subsumed by Windows. Windows 98 and Windows ME provided further improvements, but still employed the same model.

It had been apparent to both IBM and Microsoft that MS-DOS left much to be desired as an operating system, so in 1985 they jointly embarked on a project to produce a better operating system, to be called OS/2. For a variety of reasons, Microsoft dropped out of the effort in 1990. IBM continued on for a few years, but eventually discontinued it.

In its stead, Microsoft began its Windows NT project (among the various possibilities for what "NT" stands for is "new technology"). The leader of this effort was David Cutler, one of the principal architects of VMS (for the VAX-11 series) while at DEC. He modeled much of the new operating system on VMS, finally bringing the advantages of a modern operating system to Microsoft and, eventually, to most of the world. The first version of Windows NT, 3.1 (a number

apparently chosen so as not to seem inferior to the current version of “normal” Windows, also 3.1 at the time) was released in 1993. Windows ME was the last of the original Windows line; after it, starting with Windows XP, there was only one Windows operating-system family, all based on Windows NT. (Russinovich and Solomon 2005) describes the design of the Windows operating system.

1.2.4.4 Unix Continues

By the late 1980s it had become apparent to much of the computer industry, particularly those whose products ran Unix, that Microsoft was beginning to dominate. Though Unix had clearly had a lot of success, different versions of it had proliferated, all of them subtly or not so subtly incompatible with the others. For example, just to name a few, there was System V Release 3, SunOS, IRIX, Xenix, Ultrix, HP-UX, AIX, and so forth. One effort to come up with a common Unix version was already mentioned — System V Release 4 (henceforth SVR4). The only major company to adopt it at the time was Sun, who was also a major contributor to it. A number of other companies formed the OSF (Open Software Foundation⁹) consortium to develop a common code base for a standard version of Unix. This system was to be called OSF/1 and, initially, was to be based on IBM’s version of Unix, known as AIX.

After a short period of development, it was decided to base the OSF/1 kernel not on AIX, but on a combination of the Mach microkernel and 4.3 BSD. Mach was a research project at Carnegie Mellon University aimed at developing a very small operating-system kernel with minimal functionality, but capable of providing a great deal of functionality via user-level applications. At the time OSF picked it up, the additional functionality still had to reside in the kernel along with Mach, and this additional functionality was provided by 4.3 BSD. OSF/1 version 1 was released in 1991. It did not perform well. It was not until release 1.2 in 1993 that member companies actually used it in successful products. DEC replaced their Ultrix version of Unix with OSF/1, first calling it Digital Unix, then Tru64 Unix (since they had adapted it to support 64-bit architectures). IBM used it as the basis of AIX/ESA, which ran on mainframe computers.

All the operating systems discussed so far were owned in some sense by some corporation, which charged license fees for their use and usually imposed a large fee for source code. One of the reasons for Unix’s success is that its fees for source code were relatively small, particularly for universities, and that derivative products could be developed, as long as license fees were paid. The notion of an operating system that was both completely free and useful was unheard of.

In 1987 Andrew Tanenbaum of Vrije Universiteit in Amsterdam included in his operating systems textbook (Tanenbaum 1987) the complete source code for a toy operating system called Minix, based roughly on Unix and a good pedagogical tool. In 1991, Linus Torvalds, a student in Finland, bought an Intel 386 PC. MS-DOS didn’t support all the PC’s features, such as memory protection and multitasking, so he ported Minix to it and added a number of features. He called his system Linux. Eventually, it was fully compliant with pretty much all specifications for Unix developed by standards organizations such as POSIX.

Linux 1.0 (released in 1994) was ported to the DEC Alpha processor and the Sun Sparc processor by 1995. In 1996 Linux 2.0 was released. By 1998 a number of major companies, including IBM and Compaq (who had acquired DEC), announced their support for Linux. In 2000 Linux 2.2 was released and IBM announced a commitment to Linux on its servers. In 2001 Linux 2.4 was released, and Linux 2.6 was released in 2004.

⁹Sometimes glossed as “oppose Sun forever.”

In 2003 SCO,¹⁰ which had acquired the rights to Unix, sued IBM, claiming that SCO's code was in Linux and thus SCO was owed royalties. (Suing someone with lots of money may be a good idea. Suing someone who employs really good lawyers probably isn't.) In August 2007 the judge ruled that SCO was not the rightful owner of the Unix copyright, Novell is. Novell then stated that there is no Unix source code in Linux.

Two other free versions of Unix also began development in the early 1990s (and continue to be developed): FreeBSD and NetBSD. Both are derived from the BSD project at Berkeley and are highly thought of, but neither has achieved the success of Linux.

Apple, in the meantime, was still selling a system whose operating system was derived from the original MacOS. Finally, in 2000, they released MacOS X, a Unix system. Its implementation was similar to that of OSF/1, combining both Mach and BSD Unix (in the form of NetBSD and FreeBSD) in the kernel. Apple finally had an operating system that supports virtual memory and multitasking and provides a protected file system, something it had almost achieved with Lisa OS in the early 1980s. Apple's version of Unix is now the prevalent Unix operating system.

1.3 A SIMPLE OS

In this section we examine the abstractions provided by a relatively simple operating system and delve a bit into how they are implemented. Choosing an operating system, even one to discuss in a course, is fraught with controversy. What we discuss here is an early version of Unix, as it existed in the early 1970s. We choose this partly because of its simplicity and elegance, partly because (as we saw above) it had a major influence on modern operating systems, such as modern versions of Unix (Solaris, Linux, MacOS X, etc.) as well as Windows, and primarily because it is the earliest operating system whose later versions are still in common use. (Although the descendants of IBM's OS/360 are still in use, few students in operating systems courses have direct access to them.) We call it simply Unix for now, though strictly speaking it is *Sixth-Edition Unix* (see Section 1.2.3 for why it is called this).

1.3.1 OS STRUCTURE

Many early operating systems were amazingly small: Sixth-Edition Unix, for instance, had to fit into 64 KB of memory. Thus it made reasonable sense to structure it as a single executable in which all parts are stored as a single file from which they are loaded into the computer's memory when it boots. This sort of structuring is known as the *monolithic approach*. As sketched in Figure 1.1, application programs call upon the operating system via *traps*; external devices, such as disks and clocks, call upon it via *interrupts*.

Almost all computers have at least two modes of execution, *user mode* (with the fewest privileges) and *privileged mode* (with the most). To limit the damage that errant programs can do to other programs and the system as a whole, the only code that runs in privileged mode is that which is part of the operating system. In simple systems such as Sixth-Edition Unix, we generally think of the whole operating system as running in privileged mode. Everything else is an application and runs in user mode. In other systems, such as modern Windows, major subsystems providing operating-system functionality run in user mode. We discuss this in Chapter 4.

We often use the word *kernel*, as in "operating-system kernel." This generally means that portion of the operating system that runs in privileged mode, but sometimes it means a subset of this — some relatively small, key portion of the privileged-mode operating-system code. We will try to make it clear which definition we're using.

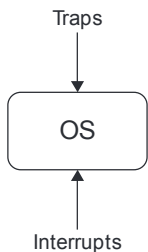


FIGURE 1.1 Simple OS structure.

¹⁰ SCO nominally stands for "Santa Cruz Operation," but the original company of that name sold its Unix business to Caldera. SCO then changed its name to Tarantella, the name of the product it was now focusing on. Caldera subsequently changed its name to SCO.

1.3.1.1 Traps

Traps are the general means for invoking the kernel from user code. We usually think of a trap as an unintended request for kernel service, say that caused by a programming error such as using a bad address or dividing by zero. However, for *system calls*, an important special kind of trap discussed below, user code intentionally invokes the kernel.

Traps always elicit some sort of response. For page faults, the operating system determines the status of the faulted page and takes appropriate action (such as fetching it from secondary storage). For programming errors, what happens depends upon what the program has previously set up. If nothing, then the program is immediately terminated. A program may establish a handler to be invoked in response to the error; the handler might clean up after the error and then terminate the process, or perhaps perform some sort of corrective action and continue with normal execution.

The response to faults caused by errors is dealt with in Unix via a mechanism called *signals* (which is also used in response to other actions; see Chapter 2). Signals allow the kernel to invoke code that's part of the user program, a mechanism known as an *upcall*.

1.3.1.2 System Calls

We've already discussed the multiple layers of abstraction used in all systems. For layers implemented strictly in user code, the actual invocation of functionality within a layer is straightforward: a simple procedure call or the like. But invocation of operating-system functionality in the kernel is more complex. Since the operating system has control over everything, we need to be careful about how it is invoked. What Unix and most other operating systems do is to provide a relatively small number of *system calls* through which user code accesses the kernel. This way any necessary checking on whether the request should be permitted can be done at just these points.

A typical example of a system call is the one used to send data to a file, the *write* system call:

```
if (write(FileDescriptor, BufferAddress, BufferLength) == -1) {
    /* an error has occurred: do something appropriate */
    printf("error: %d\n", errno) /* print error message */
}
```

Here we call *write* to request the operating system to write data to the file (we discuss later the meaning of the parameters). If the call fails for some reason, *write* returns -1 , and an integer identifying the cause of the error is stored in the global variable *errno*. Otherwise it returns a non-negative value: the number of bytes that were actually written to the file.

How *write* actually invokes the operating-system kernel depends on the underlying hardware. What typically happens is that *write* itself is a normal procedure that contains a special machine instruction causing a trap. This trap transfers control to the kernel, which then figures out why it was invoked and proceeds to handle the invocation.

1.3.1.3 Interrupts

An *interrupt* is a request from an external device for a response from the processor. We discuss the mechanism in more detail in Chapter 3. Unlike a trap, which is handled as part of the program that caused it (though within the operating system in privileged mode), an interrupt is handled independently of any user program.

For example, a trap caused by dividing by zero is considered an action of the currently running program; any response directly affects that program. But the response to an interrupt

from a disk controller may or may not have an indirect effect on the currently running program and definitely has no direct effect (other than slowing it down a bit as the processor deals with the interrupt).

1.3.2 PROCESSES, ADDRESS SPACES, AND THREADS

Probably the most important abstraction from the programmer's point of view is the *process*. We think of it both as an abstraction of memory — as an *address space* — and as the abstraction of one or more processors — as *threads* (or *threads of control*). The term “address space” covers both the set of all addresses that a program can generate and the storage associated with these addresses. In modern operating systems, address spaces are usually disjoint (they are always disjoint in Sixth-Edition Unix): processes generally have no direct access to one another's address spaces. How this is made to work has to do with address-translation hardware and its operating-system support, subjects we discuss in Chapter 7.

A single thread per process provides a straightforward programming model and was all that most operating systems supported until the early 1990s. We cover multithreaded processes in considerable detail in Chapter 2, but for now we use the simple model of single-threaded processes.

Note that the meaning of the term “process” has evolved over the years. The term originally meant the same thing as the current meaning of “thread” — see (Dennis and Van Horn 1966), who use the term “computation” to refer to what we now mean by “process.” Though some authors still use “process” in its original sense, few if any operating systems do.

What else is there to a process? To get an idea, consider the following simple C program that implements the “Sieve of Eratosthenes” to compute the first one hundred primes. In its current form it's not very useful since, after computing these primes, it immediately terminates without doing anything with them. We'll make it more useful later.

```
const int nprimes = 100;
int prime[nprimes];
int main() {
    int i;
    int current = 2;
    prime[0] = current;
    for (i=1; i<nprimes; i++) {
        int j;
        NewCandidate:
        current++;
        for (j=0; prime[j]*prime[j] <= current; j++) {
            if (current % prime[j] == 0)
                goto NewCandidate;
        }
        prime[i] = current;
    }
    return(0);
}
```

Our concern here is not prime numbers, but what the operating system must do to make this program work. The program is compiled and linked (we explain linking in Chapter 3) and

stored in a file in the file system. When we run the program, a process is created and the program is loaded from the file into the process's address space. The process's single thread of control then executes the program's code.

But how is the address space organized? The program consists of executable code and data. The code, once loaded, is never modified. Since much of the data, on the other hand, can be modified, it makes sense to segregate the two, putting the code in a special region of the address space that's protected from modification. We could simply put all the data in another readable and writable region, but we need to consider other issues too.

The variables *nprimes* and *prime* are both global, while *i*, *j*, and *current* are local. We know that the scope of global variables is the entire program, while the scope of local variables is just the block (delineated by curly braces in C) that contains them. In other words, the "lifetime" of global variables is the same as the lifetime of the program, while the lifetime of a local variable is only from when the thread enters its block to when it exits. So, we must set things up so that the portion of the address space allocated for global variables remains allocated for the lifetime of the program, but that portion allocated for a local variable remains allocated only while the thread is in the variable's scope.

Thus when the program is loaded into the address space, we'll permanently allocate space for the global variables, just beyond the space allocated for code. But there's another useful distinction to make: *nprimes* has an initial value of 100, but *prime* has no initial value, though C semantics states that its initial value is thus zero. If we group all such uninitialized variables together, we can represent them efficiently in the copy of the program stored in the file system by simply stating that we have *n* bytes of uninitialized data, which will actually be filled with zeros. For many programs, this will save a lot of space. We of course have to instantiate these variables when we load them into the address space, but there are ways to optimize this instantiation (we discuss them in Chapter 7).

The local variables can be allocated efficiently by use of a *run-time stack*: each time our thread enters a new block, it pushes a frame on the stack containing space for local variables and perhaps procedure-linkage information. Such frames are popped off the stack when the thread exits the block. So what we'll do is set up a region of the address space for the stack to reside. On most architectures, stacks range from high memory addresses to low memory addresses and thus stacks typically grow downwards.

Unix structures the address space as shown in Figure 1.2. The executable code, known as *text*, occupies the lower-addressed regions. The initialized data, known simply as *data*, follows the text. The uninitialized data is known, cryptically, as BSS (for "block started by symbol," a mnemonic from an ancient IBM 704 assembler) and comes next, followed by a dynamic region that we explain shortly. Then comes a large hole in the address space and finally a region, starting at the top and growing downwards, containing the *stack*.

Let's now modify the program a bit so that the number of primes we want to compute is passed as a parameter. Space is allocated for the primes dynamically, based on this parameter.

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
    int i;
    int current = 2;
    nprimes = atoi(argv[1]);
    prime = (int *)malloc(nprimes*sizeof(int));
    prime[0] = current;
    for (i=1; i<nprimes; i++) {
        int j;
```

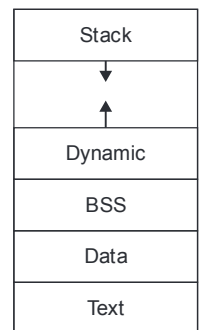


FIGURE 1.2 Unix address space.

```

NewCandidate:
    current++;
    for (j=0; prime[j]*prime[j] <= current; j++) {
        if (current % prime[j] == 0)
            goto NewCandidate;
    }
    prime[i] = current;
}
return(0);
}

```

For readers more accustomed to Java than C or C++, things are now starting to look a bit ugly. The two arguments passed to `main`, `argc` and `argv`, are part of the standard C convention for passing command-line arguments to the `main` procedure. The first procedural argument, `argc`, indicates how many command-line arguments there are. There's always at least one such argument: the name of the program. The second procedural argument, `argv`, is a vector of strings, each of which is one of the command-line arguments. How we actually get the command-line arguments to the program is discussed in Section 1.3.4.

To invoke the *primes* program with an argument of 300, we issue the following command:

```
% primes 300
```

This causes our program *primes* to be loaded into a process and the associated thread to call the *main* procedure. The `argv` vector holds two values, “primes” and “300”. Note that `argv[1]` is a character string and not an integer, so we convert it into an integer via a call to *atoi*. After doing so, *nprimes* contains the number of primes we want, so our next step is to allocate memory to hold these primes. This is done in C via a call to *malloc*, passing it the number of bytes that must be allocated. Since *prime* is an array of integers, we ask for sufficient storage to hold *nprimes* integers, where “*sizeof(int)*” returns the number of bytes in an integer.

The Java programmers will really cringe at this point, but *malloc* returns a pointer to the (typeless) block of storage allocated, which we assign to the variable *prime*. This now can be used as an array (i.e., we can refer to *prime[i]*). There is of course no bounds checking in C, so we must be careful that we don't attempt to refer to portions of *prime* that haven't actually been allocated.

Where all this leads us is to the question: where in the address space is the storage for *prime* allocated? More generally, where do storage-allocation routines such as *malloc* get their storage? The answer is: in that region of the address space called the *dynamic region* (see Figure 1.2). Its current uppermost extent is called the process's *breakpoint*. Allocation routines such as *malloc* and C++'s *new* invoke the operating system (via the system call *sbrk*) to move the breakpoint to higher addresses when necessary.

1.3.3 MANAGING PROCESSES

We've just discussed how a process is structured for running a program. How do we create a process in the first place? The only way to do so in Unix is a deceptively simple system call known as *fork*. It takes no arguments, but creates a new process that is an exact copy of the process whose thread called it (see Figure 1.3). How this copy is actually made affects performance.

In Chapter 7, we discuss some magic tricks for avoiding much of this copying. For now, we're concerned strictly with *fork*'s semantics.

Since *fork* has the effect of creating a new process containing its own thread, we need to specify where this new thread starts its execution. What happens seems a bit strange at first glance, but actually makes a lot of sense. Calls to *fork* actually return twice: once in the parent process and once in the child process.

The address space of the child contains copies of the parent's text, BSS, data, dynamic region, and stack. However, since the text is read-only, it's actually shared between the parent and child — no copying is necessary. The result of all this is a rather odd programming practice:

```
if (fork() == 0) {
    /* the child process's thread executes here */
} else {
    /* the parent process's thread executes here */
}
```

So that the two processes can figure out which ones they are, *fork* returns something different in each: it returns zero in the child and the child's process ID (a positive value known as *PID*) in the parent. The PID and other information about a process maintained by the operating system are kept in a data structure called a *process control block (PCB)*. Figure 1.4 shows the process control blocks of three processes.

For a process to terminate, it simply invokes the *exit* system call, passing an argument (constrained to be a nonnegative value less than 256, i.e., fitting in eight bits) that is its *return code*. The convention is that a process terminating successfully produces a zero return code, and positive return codes indicate some sort of problem. If a process simply returns from *main* (which is supposed to be an integer procedure and thus returns an integer), things still work: *main* returns to code that then calls *exit*, passing it the value returned from *main*. If no value is returned from *main*, *exit* is called anyway, but with an undefined (garbage) return code.

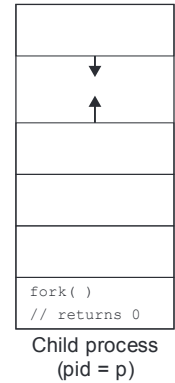
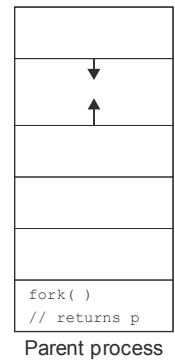


FIGURE 1.3 The effect of *fork*

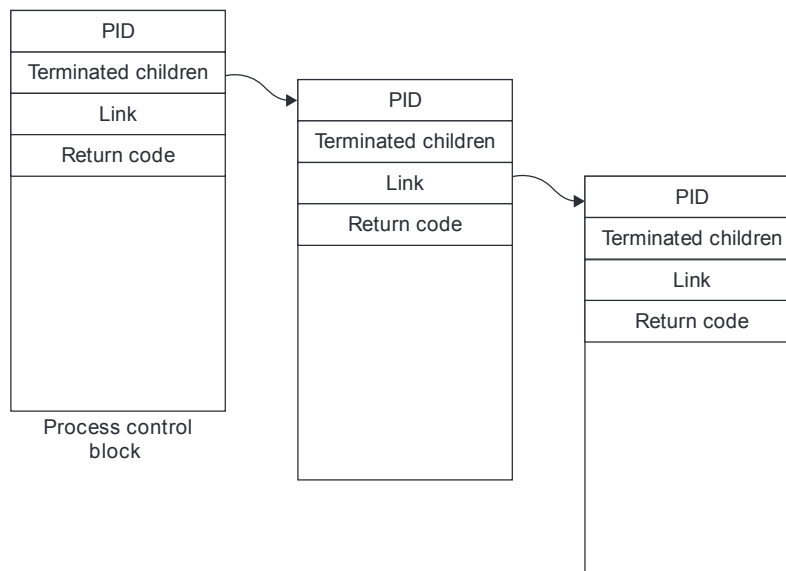


FIGURE 1.4 Process control blocks of a parent process (on the left) and two of its terminated children

Since a process's return code actually means something, it's important for other processes to find out what it is. This is done via the *wait* system call, which allows a parent process to wait until one of its children terminates:

```
short pid;
if ((pid = fork()) == 0) {
    /* some code is here for the child to execute */
    exit(n);
} else {
    int ReturnCode;
    while(pid != wait(&ReturnCode))
        ;
    /* the child has terminated with ReturnCode as its
       return code */
}
```

Here the parent process creates a child, then waits for it to terminate. Note that *wait* returns the process ID of the child that's terminated, which might not be the one most recently created (the parent might have created others). Thus it calls *wait* repeatedly until the child it's interested in terminates. The *wait* call returns the child process's return code via its argument, which points to storage provided by the caller.

A process can wait only for its children to terminate: it has no means for waiting for the termination of other processes. This implies that the operating system must keep track of the parent-child relationships among processes.

While a process is waiting for a child to terminate, it's said to be in the "sleeping state": it won't be able to execute any further instructions until a terminating child wakes it up.

The act of termination is a bit tricky. One concern is the process ID. These are 16-bit values and thus must occasionally be reused. Suppose that when a process terminates (i.e., calls *exit*), its ID is immediately made available for assignment to new processes. It might happen that before the process's parent calls *wait*, the process ID is actually assigned to a new process. Thus there could be some ambiguity about which process is being referred to by the ID.

Another termination concern is the return code: where is it stored between the moments when a process terminates and when the code is picked up by the parent via *wait*? If all storage associated with the process is released on termination, we could have a problem.

These concerns are handled as follows. When a process calls *exit* it doesn't completely disappear, but goes into what's called the *zombie state*: it's no longer active and its address space can be relinquished, but its process ID and return value are preserved in the operating system. Thus the process still exists, though the only meaningful data associated with it are its ID and return value. When the parent eventually calls *wait*, these values are finally released and all traces of the process disappear.

But what happens if the parent terminates before the child? This could mean that, since the parent is no longer around to perform the *wait*, the child will remain forever a zombie. To deal with this problem, process number 1 (the process whose ID is 1 and which is the ancestor of all other processes with greater IDs) inherits the children (including zombies) of terminated processes. It executes a loop, continually calling *wait* and finishing the termination of all its (step) children.

As shown in Figure 1.4, a process's return code is kept in its process control block. Nothing is of course stored there until the process terminates. But when the process does terminate, its return code is copied to the PCB, which is linked to its parent's queue of terminated children. By

executing the *wait* system call, the parent selects the first process from this queue, returning both the PID and return-code fields.

1.3.4 LOADING PROGRAMS INTO PROCESSES

We’ve briefly discussed setting up an address space to contain a program; now let’s look at the user-level mechanism for doing so. A family of system calls known as *exec* is provided for this. *Execs* are typically used shortly after *fork* creates a new process to replace the program with a new one. Here’s an example of their use:

```
int pid;
if ((pid = fork()) == 0) {
    /* we'll soon discuss what might take place before exec
       is called */
    execl("/home/twd/bin/primes", "primes", "300", 0);
    exit(1);
}

/* parent continues here */

while(pid != wait(0)) /* ignore the return code */
    ;
```

Here we call *execl*, which takes a variable number of arguments — the “command-line” arguments mentioned above — and passes them to the program. The first argument is the name of the file containing the program to be loaded. The second argument is the name of the program (while this seems a bit redundant, it allows one program to do different things depending on the name by which it is called). The remaining arguments are the remaining command-line arguments, terminating with 0.

The effect of the *exec* call, as shown in Figure 1.5, is to replace the entire contents of the current process’s address space with the new program: the text region is replaced with the text of *primes*. The BSS and data regions are replaced with those supplied by *primes*. The dynamic area is replaced with an empty region. The stack is replaced with the arguments that are to be passed to *main*: the total number of arguments (two in this case), followed by a vector referring to their values (“primes” and “300”). The process’s thread continues execution by calling a special start routine (loaded with the *primes* program), which then calls *main* (so that on return from *main* it calls *exit*).

Note that since the prior contents of the address space are removed, there is no return from a successful call to *exec*: there’s nothing to return to! However, if *exec* does return, then (since the prior contents clearly weren’t removed) there must have been an error. What we do in such a case is simply call *exit* with a return code indicating an error. (This is certainly not a very helpful response, but our focus for the moment is on successful outcomes.)

The parent process waits until the child has completed (calling *wait* with an argument of zero means that the caller does not want the return code). The above code fragment shows what a command shell does in response to the command:

```
% primes 300
```

In other words, it creates a new process to run the command and waits for that process to terminate.



FIGURE 1.5 The effect of *exec*.

1.3.5 FILES

As we've pointed out, our *primes* example isn't very useful since it doesn't leave the results of its computation where others (programs or people) can use them. What's needed is access to someplace outside the process that's shared with others. The notion of a *file* is our Unix system's sole abstraction for this concept of "someplace outside the process" (modern Unix systems have additional abstractions). Unix uses files both as the abstraction of persistent data storage (such as on disks) and also as the means for fetching and storing data outside a process, whether that data is stored on disk, in another process, or in some other device, such as a keyboard or display.

1.3.5.1 Naming Files

For our current discussion we're concerned about both how to refer to such "places" outside the process and how programs transfer data to and from such places. Since the place is outside the process, we need a different space from the process's address space. The nature of such spaces was an issue a number of decades ago, but pretty much all systems today use tree-structured directory systems for naming files and similar objects. These should be familiar to everyone with enough computer experience to have gotten this far in this text: a file is named by stringing together the names assigned to the edges forming a path from the root of the tree to the file.

Unix uses forward slashes as separators between the names; Windows uses back slashes. That the path starts at the root is indicated by starting the name with the separators. Such path names generally have the beginning (such as the root) at the left, though the Internet's naming scheme (Domain Name System — DNS) has it on the right.

The name space provided by the directory system is generally shared by all processes running on a computer (and perhaps by all processes running on a number of computers). Unix provides a means to restrict a process to a subtree: one simply redefines what "root" means for the process. Thus files are identified by their path names in the directory system.

Since the directory-system name space is outside the process, special means are required to access it. The usual model is that one provides the name of the desired file to the operating system, and the operating system returns a *handle* to be used to access the file. What's going on behind the scenes is that the operating system, somewhat laboriously, follows the path provided by the name, checking to make certain that the process is allowed appropriate access along the path. The returned handle provides a direct reference to the file so that such expensive path-following and access verification isn't required on subsequent accesses.

This use of a handle to refer to an object managed by the kernel is fairly important. We'll later see it generalized to the notion of a *capability* (Chapter 8). In abstract terms, possession of a handle gives the holder not only a reference to an object in the kernel, but also certain limited rights for using the object. In the case of files, as we discuss, a handle allows the holder to perform certain actions on a file.

The following code uses the *open* system call to obtain a file's handle, then uses the handle to read the file:

```
int fd;
char buffer[1024];
int count;
if ((fd = open("/home/twd/file", O_RDWR) == -1) {
    /* the file couldn't be opened */
    perror("/home/twd/file");
    exit(1);
}
```

```

if ((count = read(fd, buffer, 1024)) == -1) {
    /* the read failed */
    perror("read");
    exit(1);
}
/* buffer now contains count bytes read from the file */

```

Here we use the *open* system call to access the directory name space to get a handle for the file whose path name is “/home/twd/file”. We’ve indicated by the second argument that we want both read and write access to the file: if for some reason such access is not permitted, the *open* call fails. If it succeeds, *fd* contains the handle (known in Unix as a *file descriptor*) for the file.

We then use this handle as an argument to the *read* system call to identify the file and attempt to transfer its first 1024 bytes into *buffer*. *Read* returns the number of bytes that were actually transferred: it could be less than what was asked for because, for example, the file might not be that large. The *perror* routine prints, to file descriptor 2, its argument followed by a message explaining the error number currently in the global variable *errno* (recall that that’s how failing system calls leave the error number).

1.3.5.2 Using File Descriptors

These handles (or *file descriptors*) form what is essentially an extension to the process’s address space, allowing the process unhindered access to the associated files. This address-space extension survives *execs*. Thus files open before an *exec* takes place are open afterwards. This property is exploited as a way to pass the open files as additional parameters to the *exec*. File descriptors are small nonnegative integers that are allocated lowest-available-number first. By convention, programs expect to read their primary input from file descriptor 0, to write their primary output to file descriptor 1, and to write error messages to file descriptor 2. By default, input from file descriptor 0 comes from the keyboard and output to file descriptors 1 and 2 goes to the display (or current window).

However, as shown in the following code, different associations can be established in a process before an *exec*:

```

if (fork() == 0) {
    /* set up file descriptor 1 in the child process */
    close(1);
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        perror("/home/twd/Output");
        exit(1);
    }
    execl("/home/twd/bin/primes", "primes", "300", 0);
    exit(1);
}

/* parent continues here */

while(pid != wait(0)) /* ignore the return code */
    ;

```

Once the new process is created, we close file descriptor 1, thereby removing it from the extended address space. Then we open a file; since file descriptors are allocated lowest first, the file is assigned file descriptor 1. Thus after the *exec*, the *primes* program finds the new file associated with file descriptor 1.

As it stands, our primes program doesn't use any files, so let's modify it so it does:

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
    int i;
    int current = 2;
    nprimes = atoi(argv[1]);
    prime = (int *)malloc(nprimes*sizeof(int));
    prime[0] = current;
    for (i=1; i<nprimes; i++) {
        int j;
NewCandidate:
        current++;
        for (j=0; prime[j]*prime[j] <= current; j++) {
            if (current % prime[j] == 0)
                goto NewCandidate;
        }
        prime[i] = current;
    }
    if (write(1, prime, nprimes*sizeof(int)) == -1) {
        perror("primes output");
        exit(1);
    }
    return(0);
}
```

Now the computed set of primes, which will appear in */home/twd/Output*, can be used by others.

However, this output wouldn't be all that useful if it were written to the display, rather than the named file — the numbers are in binary, rather than expressed in decimal notation as strings of characters. So, to make our program useful in both situations, let's change it so that it writes its output as strings of decimal digits.

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
    int i;
    int current = 2;
    nprimes = atoi(argv[1]);
    prime = (int *)malloc (nprimes*sizeof(int));
    prime[0] = current;
```



```

    for (i=1; i<nprimes; i++) {
        int j;
    NewCandidate:
        current++;
        for (j=0; prime[j]*prime[j] <= current; j++) {
            if (current % prime[j] == 0)
                goto NewCandidate;
        }
        prime[i] = current;
    }
    for (i=0; i<nprimes; i++) {
        printf("%d\n", prime[i]);
    }
    return(0);
}

```

The *printf* routine's first argument contains a format code, “%d”, that instructs it to convert the next argument to a string of decimal digits suitable for printing. The “\n” instructs it to append a new-line character to the end of the string. The string is written to file descriptor 1, which we've established as our file. Note that *printf* itself calls *write* to send its output to file descriptor 1.

Our program can be invoked as follows from a command shell, with the substitution of the given output file for the display:

```
% primes 300 > /home/twd/Output
```

The “>” parameter instructs the command shell to “redirect” the output to the given file. If “>” weren't there, then the output would go to the display or window.

Let's look more deeply at how file descriptors are used. They refer not just to files, but to the process's current “context” for each file. This context includes how the file is to be accessed (recall that the second parameter of the *open* system call specifies it — read-write, read-only, or write-only) as well as the current location within the file. This latter is important since the read and write system calls don't specify this — the usual case is that files are accessed sequentially. Each read of the file returns the next specified number of bytes. Each write to the file puts data in the next locations in the file.

The context information itself must be maintained by the operating system and not directly by the user program. Some of it is important operating-system information — data that's meant only for the operating system's internal purposes. For example, when one opens a file, let's say for read-only access, the operating system checks that read-only access is indeed permitted. Rather than making this check each time the user reads the file via the returned file descriptor, the system keeps the result of the check with the context information. If the user program could modify this directly, it could change it from read-only to read-write and thus be able to write to a file without suitable permission.

So, this information is stored not in the process's address space, but in the kernel's address space (and thus is not directly accessible by the user). The user simply refers to the information using the file descriptor, which is, in effect if not in reality, an index into an array maintained for the process in the kernel's address space. This insures that the process has such indirect access only to its own file information. Thus file descriptors appear as shown in Figure 1.6.

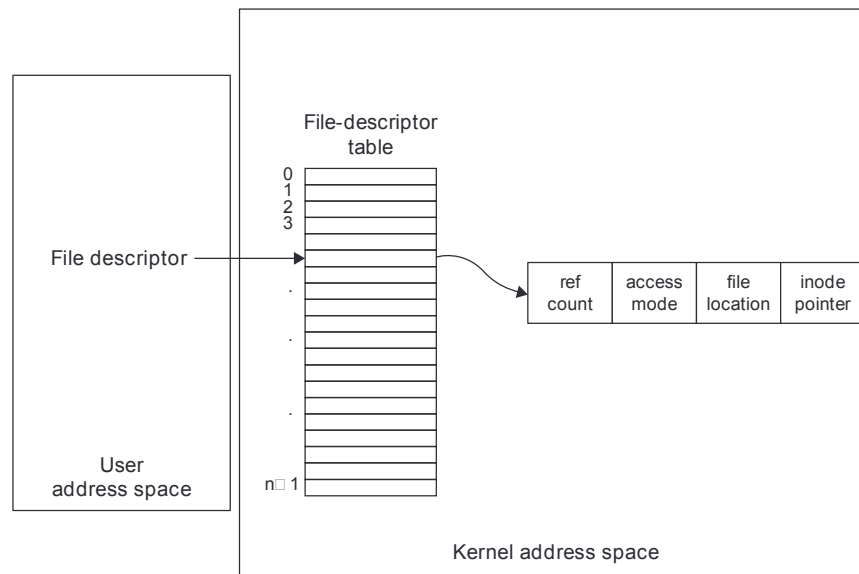


FIGURE 1.6 File-descriptor table

The context information associated with each open file is not stored directly in the file-descriptor table. For reasons explained shortly, there's another level of indirection: the file-descriptor-table entry actually points to another data structure containing this context information. This context information contains yet another pointer, this time to a data structure called an *inode* (for *index node*, a term we discuss in Chapter 6), representing the file itself.

Now we look at some file-descriptor subtleties. We know that file descriptor 1 is used for a process's normal output and file descriptor 2 for its error messages. We also know that we can redirect such output to go to specified files. Suppose we want to send both the process's normal output and its error messages to the same file, which is other than the display or current window. We might set this up with the following code, which closes file descriptors 1 and 2, then opens a file twice. Since each successful call to *open* returns the lowest available file descriptor (and since file descriptor 0 is already taken), we end up with file descriptors 1 and 2 both referring to the same file.

```
if (fork() == 0) {
    /* set up file descriptors 1 and 2 in the child process */
    close(1);
    close(2);
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        exit(1);
    }
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        exit(1);
    }
    execl("/home/twd/bin/program", "program", 0);
    exit(1);
}
/* parent continues here */
```

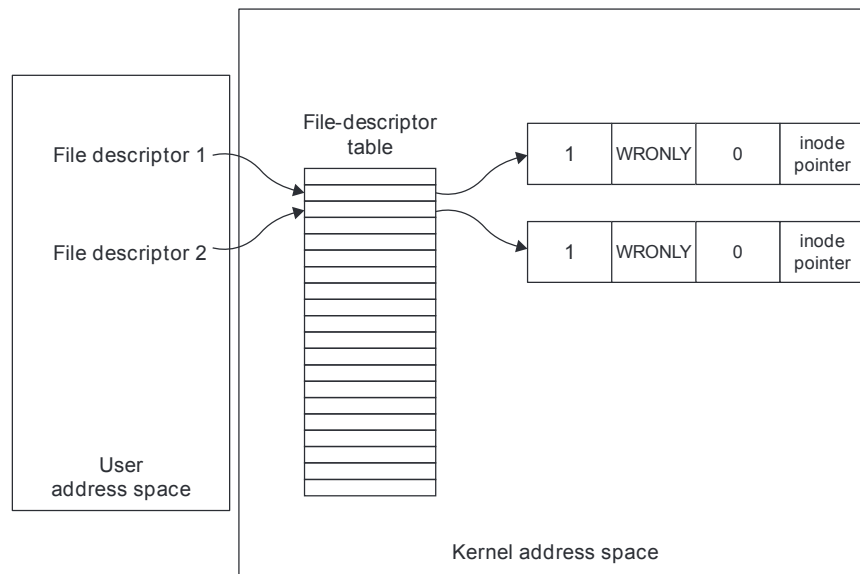


FIGURE 1.7 Redirected output

The effect of this is shown in Figure 1.7.

At this point, each of the context structures has a reference count of one (each is pointed to by exactly one file-descriptor-table entry), each has been opened write-only, each refers to the beginning of the file, and each refers to the same inode (for the same file).

Now suppose the program writes 100 bytes of normal output. The effect is shown in Figure 1.8. The context information for file descriptor 1 indicates that the next *write* will take place at location 100 in the file. But the context information for file descriptor 2 indicates that the next *write* through it will be at the beginning of the file. If the program now writes an error message (via file descriptor 2), the message will overwrite the 100 bytes of normal output that had already been written. This is probably not what's desired.

What we'd like is shown in Figure 1.9. Here both file descriptors refer to the same context information. This is accomplished with the *dup* system call:

```
if (fork() == 0) {
    /* set up file descriptors 1 and 2 in the child process */
    close(1);
    close(2);
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        exit(1);
    }
    dup(1); /* set up file descriptor 2 as a duplicate of 1 */
    execl("/home/twd/bin/program", "program", 0);
    exit(1);
}
/* parent continues here */
```

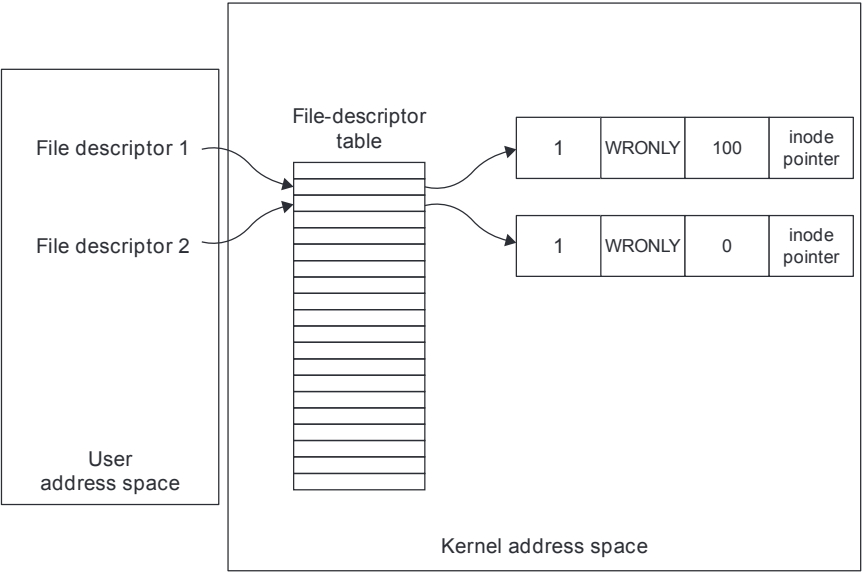


FIGURE 1.8 Redirected output after normal write

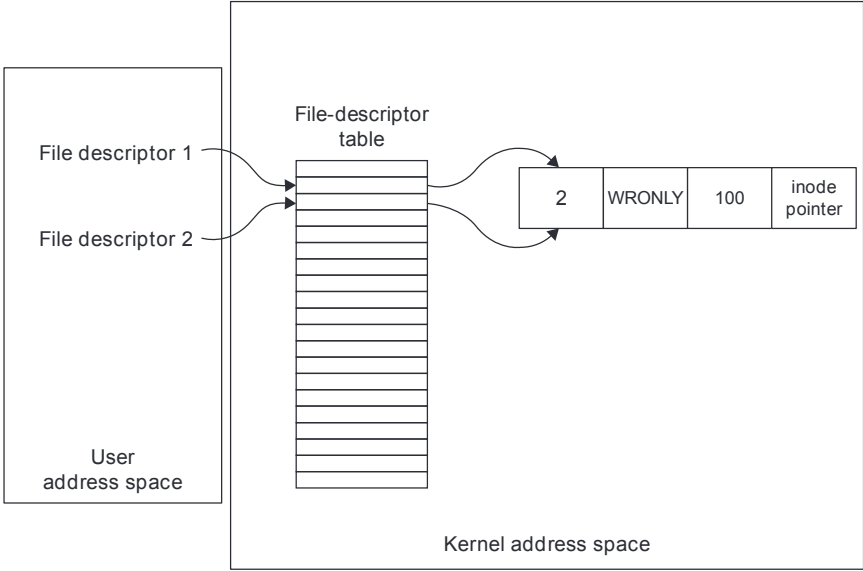


FIGURE 1.9 Redirected output with shared context information

To request such redirection using a command shell, we do the following (our example uses *csh* syntax):

```
% program >& /home/twd/Output
```

Note that in pretty much all Unix command shells, the ampersand (“&”) has another use as well. At the end of a command line it means that the parent process does not wait for the child to terminate; in other words, the parent doesn’t issue the *wait* system call after the *fork*, but goes on to the next command.

Finally, what should happen with the file descriptor and context information when a process forks? From what we know about *fork*, it makes sense for the child process to have a copy of the parent’s file-descriptor table. But should it also have a copy of each of the context-information structures? Consider the following code:

```
int logfile = open("log", O_WRONLY);
if (fork() == 0) {
    /* child process computes something, then does: */
    write(logfile, LogEntry, strlen(LogEntry));
    ...
    exit(0);
}

/* parent process computes something, then does: */

write(logfile, LogEntry, strlen(LogEntry));
...
```

The intent here is that the “log file” contain everything written to it; thus when each process writes to it, the data goes to the current end of the file. If, however, each process had its own copy of the context information for the file descriptor corresponding to *logfile*, the writes of one process would overwrite those of the other. Instead, the two processes should share the context information. Thus the setup after a *fork* is as shown in Figure 1.10.

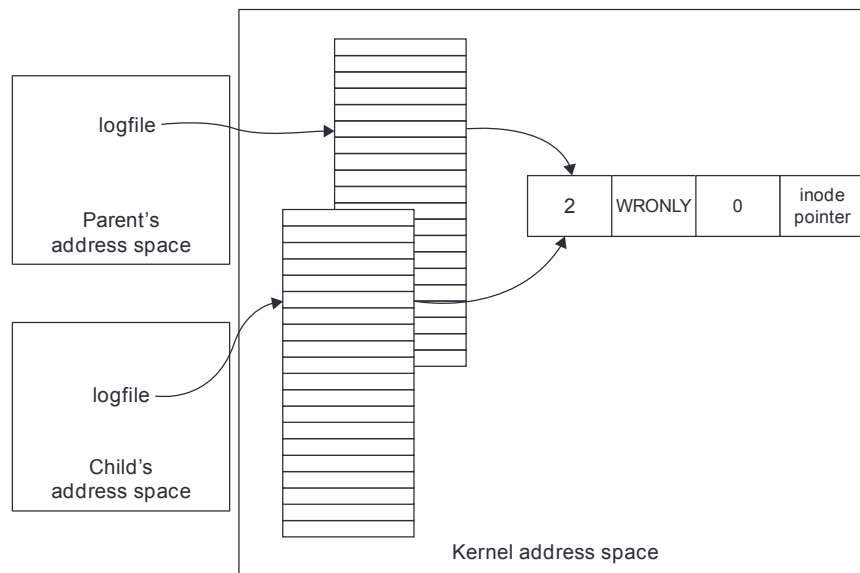


FIGURE 1.10 File descriptors and context after a *fork*

1.3.5.3 Random Access

As we've seen, the normal mode of access to files is sequential: successive reads or writes to a file are to successive locations in the file. Though this is probably what's desired in most situations, sometimes we'd like to access a file randomly, reading or writing arbitrary locations within it. This turns out to be easily done, since the *read* and *write* system calls simply look at the contents of the file-location field of the context structure and increment it after the operation.

Thus to read or write starting at an arbitrary location, all we have to do is provide a means for setting this file-location field. This is done with the *lseek* system call. The example below shows using *lseek* to print the contents of a file backwards:¹¹

```
fd = open("textfile", O_RDONLY);
/* go to last char in file */
fptr = lseek(fd, (off_t)-1, SEEK_END);
while (fptr != -1) {
    read(fd, buf, 1);
    write(1, buf, 1);
    fptr = lseek(fd, (off_t)-2, SEEK_CUR);
}
```

The first argument to *lseek* is the file descriptor, the second and third arguments (of type *off_t*, an integer type) indicate the value that goes into the file-location field. The third argument specifies where we are measuring the offset from (SEEK_SET indicates the offset is interpreted as from the beginning of the file, SEEK_CUR means from the current position in the file, and SEEK_END means from the end of the file). Note that *lseek* does no actual I/O; all it does is set the file-location field.

What we've done in the example above is to start with the file location set to point to the last character in the file. After reading a byte, the location is advanced by one, so to get the previous character, we subtract two from it.

1.3.5.4 Pipes

An interesting construct based on the file notion is the *pipe*. A pipe is a means for one process to send data to another directly, as if it were writing to a file (see Figure 1.11). The process sending data behaves as if it has a file descriptor to a file that has been opened for writing. The process receiving data behaves as if it has a file descriptor referring to a file that has been opened for reading.

A pipe and the two file descriptors referring to it are set up using the *pipe* system call. This creates a pipe object in the kernel and returns, via an output parameter, the two file descriptors that refer to it: one, set for write-only, referring to the input side and the other, set for read-only, referring to the output end. Since this pipe object, though it behaves somewhat like a file, has no name, the only way for any process to refer to it is via these two file descriptors. Thus only the process that created it and its descendants (which inherit the file descriptors) can refer to the pipe.

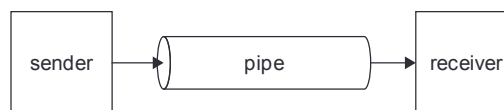


FIGURE 1.11 Communication via a pipe

¹¹ This is definitely not a good way to print a file backwards! It simply illustrates what you can do with *lseek*.

Here's a simple pipe example:

```
int p[2];    /* array to hold pipe's file descriptors */
pipe(p);    /* create a pipe; assume no errors */
/* p[0] refers to the output end of the pipe */
/* p[1] refers to the input end of the pipe */
if (fork() == 0) {
    char buf[80];
    close(p[1]);    /* not needed by the child */
    while (read(p[0], buf, 80) > 0) {
        /* use data obtained from parent */
        ...
    }
} else {
    char buf[80];
    close(p[0]);    /* not needed by the parent */
    for (;;) {
        /* prepare data for child */
        ...
        write(p[1], buf, 80);
    }
}
```

1.3.5.5 Directories

A directory is essentially a file like the others we've been discussing, except that it is interpreted by the operating system as containing references to other files (some of which may well be other directories). From a logical perspective, a directory consists of an array of pairs of *component name* and *inode number*, where the latter identifies the target file's *inode* to the operating system (recall that an inode is a data structure maintained by the operating system to represent a file).

Every directory contains two special entries, “.” and “..”. The former refers to the directory itself, the latter to the directory's parent. In Figure 1.12, the directory is the root directory and has no parent, and thus its “..” entry is a special case that refers to the directory itself.

Component name	Inode number
Directory entry	
.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

FIGURE 1.12 Sample directory

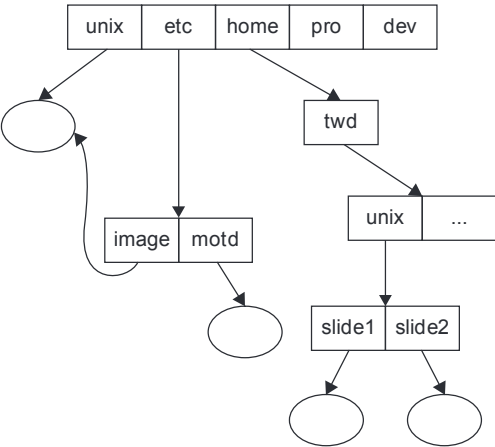


FIGURE 1.13 Directory hierarchy

.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

.	4
..	1
image	117
motd	33

FIGURE 1.14 Directory contents

We normally think of directories as forming a tree-structured hierarchy, but Unix and many other operating systems allow limited deviations from trees in the form of hard links (we’ll see the soft kind of link shortly). A *hard link* is a reference to a file in one directory that also appears in another. The only restriction is that such links may not be made to directories. Figure 1.13 and Figure 1.14 show an example. To create a hard link, one uses either the *link* system call or the *ln* shell command.

Figure 1.14, showing the logical contents of both the *root* (/) and */etc* directories, demonstrates that */unix* and */etc/image* are the same file. Note that if the directory entry */unix* is deleted (via the shell’s *rm* command), the file (represented by inode 117) continues to exist, since there is still a directory entry referring to it. However, if */etc/image* is also deleted, then the file has no more links and is removed. To implement this, the file’s inode contains a link count indicating the total number of directory entries that refer to it. A file is actually deleted only when its inode’s link count reaches zero.

Note: suppose a file is open, i.e., is being used by some process, when its link count becomes zero. The file cannot be deleted while the process is using it, but must continue to exist until no process has it open. Thus the inode also contains a reference count indicating how many times it is open: in particular, how many system file table entries point to it. A file is deleted when (and only when) both the link count and this reference count become zero. The shell’s *rm* command is implemented using the *unlink* system call.

A different kind of link is the *soft* or *symbolic link*, a special kind of file containing the name of another file. When the kernel processes such a file, it does not simply retrieve its contents, but replaces the portion of the directory path that it has already followed with the contents of the soft-link file and then follows the resulting path. Thus referencing */home/twd/mylink* in Figure 1.15 yields the same file as referencing */unix*. Referencing */etc/twd/unix/slide1* results in the same file as referencing */home/twd/unix/slide1*. Symbolic links are created by using the shell’s *ln* command with the “-s” flag, which is implemented using the *symlink* system call.

Each process, as part of its kernel-maintained context, has what’s called its *working directory*. A pathname that doesn’t start with “/” (i.e., it doesn’t start with the root directory) is considered to start at the process’s current working directory. This directory is set with the *cd* shell command or the *chdir* system call. You might try to figure out how the *pwd* shell command, which prints out the pathname of the current working directory, is implemented (see Exercise 13 in this chapter).

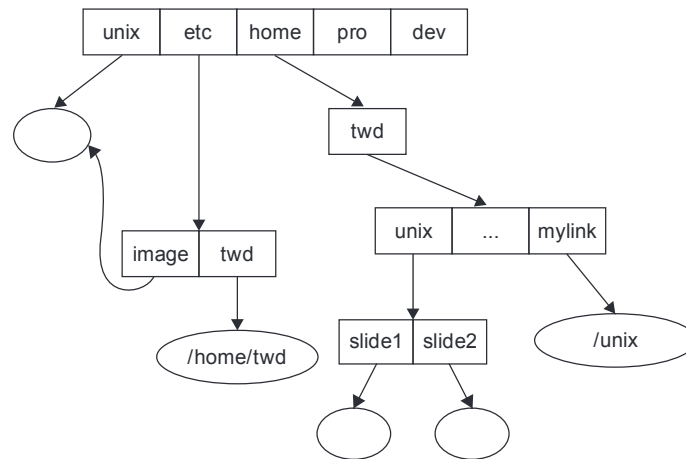


FIGURE 1.15 Symbolic links

1.3.5.6 Access Protection

Among an operating system's duties is making certain that only authorized processes are allowed access to system resources. There are various models for doing this. Unix (and many other operating systems, including Windows) associates with files (and other resources) some indication of which *security principals* are allowed access, along with what sort of access is allowed. A “security principal” is normally a user or group of users, though a “user” isn't necessarily a human but can be, say, some identity used by processes performing system functions. Each running process has potentially a number of security principals associated with it: all processes have a user identification as well as a set of group identifications (though in Sixth-Edition Unix the size of this set was exactly one).

Each file has associated with it a set of access permissions indicating, for each of three classes of principals, what sorts of operations on the file are allowed. The three classes are the owner of the file, known as *user*, the group owner of the file, known simply as *group*, and everyone else, known as *others*. The operations are grouped into the classes *read*, *write*, and *execute*, with their obvious meanings. The access permissions apply to directories as well as to ordinary files, though the meaning of *execute* for directories is not quite so obvious: one must have *execute* permission for a directory in order to follow a path through it.

The system, when checking permissions, first determines the smallest class of principals the requester belongs to: user (smallest), group, or others (largest). It then checks for appropriate permissions within the chosen class.

The permissions associated with a file are set when the file is created (see Section 1.3.5.7) and may be modified using the *chmod* system call or shell command. Consider the following example. Here we use the “ls” command to list the contents of a directory and its subdirectories, showing access permissions.

What this output says is that the current directory (indicated as “.”) contains two subdirectories, *A* and *B*, each containing a file or two, all with access permissions as shown. Note that the permissions are given as a string of characters: the first character indicates whether or not the file is a directory, the next three characters are the permissions for the owner of the file, the next three are the permissions for the members of the file's group, and the last three are the permissions for the rest of the world. Also shown is the name of the file's user association (tom or trina in the example), the group association (adm in all cases), the size of the

```
% ls -lR
.:
total 2
drwxr-x--x  2 tom    adm    1024 Dec 17 13:34 A
drwxr----- 2 tom    adm    1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 tom    adm     593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 tom    adm     446 Dec 17 13:34 x
-rw----rw-  1 trina  adm     446 Dec 17 13:45 y
```

file in bytes, the time it was last modified, and the name of the file (the last component of the pathname).

Suppose users *tom* and *trina* are members of the *adm* group but *andy* is not. Consider the following access questions:

May *andy* list the contents of directory *A*?

May *andy* read *A/x*?

May *trina* list the contents of directory *B*?

May *trina* modify *B/y*?

May *tom* modify *B/x*?

May *tom* read *B/y*?

The answers are

1. No. Since *andy* fits in the category of “others” for directory *A*, his permissions for that directory are execute only. This means that though he’s allowed to follow a path through that directory, he’s not allowed to list its contents.
2. Yes. As mentioned above, *andy* may follow a path through directory *A*. Since he has read permission for file *x* within *A*, he may read *A/x*.
3. Yes. Since *trina* is a member of the *adm* group, *B*’s group association is *adm*, and the group has read permission, *trina* may list *B*’s contents.
4. No. Though *trina* does have read and write permission for file *y*, she cannot follow a path through *B* into it. Thus, since she can’t get to *B/y*, she may not modify it.
5. No. Although *tom* may follow the path through *B* to *x*, *x* has a group association of *adm*, *adm* is allowed read-write access to *x*, and *tom* is a member of *adm*: since *tom* is listed as the user (or owner) of *x*, with only read permission for the file, *tom* may not modify the file.
6. No. Since *tom* is a member of *adm*, his permissions are given by the group category and not the others category. Since the group has no permissions, *tom* may not read the file.

1.3.5.7 Creating Files

One can create a file in Unix by using the *creat*¹² system call. This was the only way to create a file in Sixth-Edition Unix, but a separate `O_CREAT` flag was later given to *open* so it, too, can be used to create files. The *creat* system call fails if the file already exists. For *open*, what happens if the file already exists depends upon the use of the flags `O_EXCL` and `O_TRUNC`. If `O_EXCL` is included with the flags (e.g., *open*("newfile", `O_CREAT|O_EXCL`, 0777)), then, as with *creat*, the call fails if the file exists. Otherwise, the call succeeds and the (existing) file is opened. If `O_TRUNC` is included in the flags, then, if the file exists, its previous contents are eliminated and the file (whose size is now zero) is opened.

When a file is created by either *open* or *creat*, the file's initial access permissions are the bitwise AND of the mode parameter and the complement of the process's *umask*, a value that's part of each process's context and maintained by the operating system.

At first glance, this seems unnecessarily complicated, but it's less so when you understand how and why it's done. The idea is that standard programs, such as compilers and editors, create files with appropriate permissions given to everyone. For example, compilers produce executable programs whose permissions are read, write, and execute for the user, group, and others. Editors produce files whose permissions are read and write for the user, group, and others. In octal notation, these permissions are 0777 and 0666 respectively (where each octal digit encodes the permissions for one category of principals).

You typically do not want your files to be, by default, accessible to everyone in the world. Thus you set your *umask* to indicate which permissions you generally want turned off. So, for example, you might decide that, by default, files you create should not be accessible by others and should have at most read access by the file's group. You'd then set your *umask* to 0037. The rightmost octal digit (being all ones in binary) indicates that all permissions are turned off for the category others. The next octal digit has the bits set for write access (4) and execute access (1), indicating that you want these permissions turned off. Thus when a compiler creates an executable file, its permissions will be $0777 \& \sim(0037)$, which is 0740.

You set the *umask* using either a system call or a shell command.

Sixth-Edition Unix was released in 1975. What has happened with operating systems since then? Unix versions proliferated over the next two decades, but only a relatively few now survive. Though there are others, we consider Solaris as the primary example of a proprietary Unix system. Microsoft's Windows family of operating systems has come to dominate the marketplace. On the other hand, Unix implementations as open-source software, in particular the Linux and BSD operating systems, have relatively recently come out of the blue to become significant competitors. Even more recently, Chromium OS¹³ is being developed as open-source software by Google, on top of a Linux base.

Our goal in this section is not to discuss the relative merits of these systems, but to introduce some of the new¹⁴ ideas and functionality implemented in them. We first look at those items

1.4 BEYOND A SIMPLE OS

¹²The story is told that someone once asked Dennis Ritchie, one of the two developers of the first Unix implementation, if he'd do anything differently if he were to do it all over again. He replied that he'd add an "e" to the end of *creat*.

¹³Since Chromium OS is built on top of Linux, isn't it really Linux and not a separate OS? Its intended use as a platform for web applications and its heavy emphasis on security stresses functionality that is not emphasized in most other Linux distributions. Though it might not be a new species of OS, it is definitely on its way to being a subspecies.

¹⁴"New," in many cases, means "not done in Sixth-Edition Unix." Many of the notions discussed here were first introduced in earlier systems, as we will point out.

that extend what was used in early Unix. Next we mention some areas that became important since the early days of Unix.

1.4.1 EXTENSIONS

The notions of process, file, directory, file descriptor, and so forth are seemingly simple, but amazingly useful. They weren't introduced by Unix, but Sixth-Edition Unix gave us an elegant example of their use. In this section we discuss how newer systems have elaborated these.

1.4.1.1 Multithreaded Processes

We introduced processes in Section 1.3.2. Most early operating systems provided only single-threaded processes; in fact, the term “process” was often used to mean “thread.” By the late 1980s it became clear that multithreaded processes, i.e., multiple threads of control sharing the same address space, are pretty useful and important. Though a standard *API*¹⁵ for threads in Unix wasn't accepted until 1995, various implementations with a variety of APIs were being used from 1985, if not earlier. Threads of some form were in use as early as the 1960s (Dennis and Van Horn 1966). We cover the use of threads in great detail starting in the next chapter and their implementation in Chapter 5.

1.4.1.2 Virtual Memory

Sixth-Edition Unix gave each process its own address space, but all of the currently running process had to fit into the computer's memory at once, along with the operating system. This was a major limitation back then, when memory was rather expensive (tens of thousands of dollars per megabyte). Virtual memory separates the process's view of memory — the address space — from physical resources. This allows us not only to give each process its own private address space as in early Unix, but also to assign physical memory to address spaces on an as-needed basis, allowing large address spaces to be used on machines with relatively modest amounts of physical memory.¹⁶ It also makes possible the many other useful tricks we discuss in Chapter 7. Though the concept of virtual memory was first used in 1960 (Kilburn, Edwards, et al. 1962), it didn't make it into Unix until 1979 (and into Windows until 1993).

1.4.1.3 Naming

A major contribution of early Unix systems was their simple design. Surprisingly, even such apparently obvious ideas as tree-structured directories for file systems weren't commonplace in the operating systems of the early 1970s. Unix's designers extended the notion of naming files by their directory-system path names to naming devices, such as terminals, printers, and telephone lines, as well.

This notion of naming pretty much everything using the directory system was extended further in the early 1980s with the introduction of the */proc* file system (Killian 1984). */proc* used the directory system to name operating-system objects such as processes, thus allowing programs to “open” these and then operate on them using file-related system calls (such as *read* and

¹⁵ API: *Application Program Interface*, a common buzzword that means the names and arguments of the procedures, functions, methods, etc. that application programs call to access the functionality of a system. This is important to programmers because knowing the API, one can write programs that use the system without concern for its implementation details. For example, *read*, *write*, *open*, *close*, *fork*, and *wait* are all part of the API for Unix. The API is different from the *Application Binary Interface (ABI)*, the machine-language instructions necessary for accessing a system's functionality. The ABI is important to providers of software in binary form, who must be sure that their code will run without recompilation or relinking.

¹⁶ In the late 1970s, the concern was 32-bit address spaces, a word size large enough to address up to 4 gigabytes of memory, when four gigabytes of memory cost millions of dollars. Today the concern is 64-bit address spaces, a word size large enough to address up to 2⁶⁴ bytes of memory, an amount of memory that again costs millions of dollars.

write). This notion was extended even further in Plan 9 (Pike, Presotto, et al. 1995), an offshoot of Unix.

1.4.1.4 Object References

As we discussed in Section 1.3.5, while names are used to identify operating-system objects such as files, a process needs some sort of handle so it can perform operations on an object. Unix has pretty much only one type of object: objects on which one can use file-related system calls.

Microsoft's Win-32 interface supports a number of different types of operating-system objects, such as:

- processes
- threads
- files
- pipes
- timers
- etc.

While these different types of objects have much in common, including some of the system calls used for operating on them, they have significant differences as well. However, they are implemented and used much as Unix's file descriptors. See Microsoft's web site for details.

1.4.1.5 Security

Security wasn't a big deal back in the mid-1970s. Few computers were networked together. One could not bring down the planet's economy by launching an attack on computers. It was a simpler era. Even so, there had already been a large amount of research on computer security. The approach taken in Unix (and outlined above), though a major advance over common practice, was simple but, as always, elegant. The primary change in modern Unix systems to the basic model is that users can be in more than one group at a time. That's it.

Though this basic model suffices for most work, it's not completely general and doesn't handle all the demands of networked computers. Basic Unix security covers file access and not much more. Many modern systems, such as recent versions of Windows and, to a lesser extent, recent versions of Unix, not only allow generalizations of the file-access model but also support various sorts of *privileges* that control what a user is allowed to do, such as shut down a machine, add new software, perform backups, etc. A major emphasis in modern operating systems is the isolation of services so that the effects of successful attacks are limited. We cover much of this in Chapter 8.

1.4.2 NEW FUNCTIONALITY

Three major things have affected operating systems since 1975:

- networking
- interactive user interfaces
- software complexity

Early Unix systems were standalone computers. To get data from one to another, you wrote it to tape, then carried the tape to the other computer. A few advanced systems could communicate by low-speed telephone lines. You communicated with computers by typing commands to

them. A major advance was the use of display terminals rather than teletypes (or punched cards). It's really amazing that people got any work done at all.

Adding support for networking, interactive user interfaces, and the myriad other items required to make all this usable has dramatically increased the amount of code that goes into an operating system. The Sixth-Edition Unix kernel was a few thousand lines of C code. Modern Unix systems are a few million lines of C code. Modern Windows systems are probably larger still. It's no longer feasible for the entire operating system to be a single program that sits in one file on disk. (In fact, the Solaris kernel is too large to fit in the largest file supported on Sixth Edition Unix.)

1.4.2.1 Networking

The typical modern computer is anything but a standalone device. It doesn't just connect to the Internet. It must support TCP/IP, the communication protocols. It looks up names, not just in directories on local disks but in DNS, the world-wide domain name service. It can access files not just locally but on any number of servers on the Internet. It can download code, malicious and otherwise, from other computers and execute it immediately. It can even arrange to execute on other computers and have the results sent back.

We cover a small amount of what's involved in supporting all this in Chapter 9. Covering the rest adequately requires much additional material beyond the scope of this book.

1.4.2.2 Interactive User Interfaces

An interactive user interface is not just another application, handled just like all the others. It's often what sells a computer. The entire system must be designed so the user interface (UI) works well. Sixth-Edition Unix didn't have to handle mouse events, bit-mapped displays, graphics processors, and so forth. It didn't step all over itself to make certain that the interactive user received excellent response. Though we don't explicitly address the support of user interfaces in this book, we do cover much of what an operating system must do to provide such support.

1.4.2.3 Software Complexity

In Sixth-Edition Unix, if you added a new device, such as a tape drive, to your system, you had to first obtain (or write) a "device driver" to handle the device itself. You then had to modify the operating system's source code, by adding references to the driver to a few tables. Then you recompiled the operating system and rebooted your computer. If you did it all correctly, the tape drive was now part of the system.

In a modern Windows system, if you buy a new device that's reasonably popular, odds are you can simply plug it into your computer, click a few items in some dialog boxes, and almost immediately your device is part of your system — even a reboot isn't necessary. Sometimes you may have to supply a driver from a CD, but it's read in, the operating system is automatically updated, and you can start using your device almost instantly.

To make all this possible, the operating system supports the dynamic linking of modules into a running system. We haven't covered enough to explain this in detail quite yet, but we will in Chapter 3.

An advantage of being able to add modules to an operating system dynamically is that fewer modules need to be in the initial operating system: they can be loaded in when needed. This actually provides only marginal reduction in the overall complexity; in fact, adding support for dynamic loading of modules can make the operating system even more complex.

Another approach, developed in the 1980s in the Mach (Rashid, Baron, et al. 1989) and Chorus (Zimmermann, Guillemont, et al. 1984) systems, is to pull much of an operating system's functionality out of its kernel and provide it via code running as normal applications. Thus the

kernel itself, now known as a *microkernel*, doesn't have to do all that much; instead, it facilitates communication with those applications that provide the needed functionality. Though promising and interesting, the approach was never a commercial success, though a variation based on the results of some of this work is the basis of Apple's Unix implementation in MacOS X. We cover microkernels in Chapter 4.

At this point you should have a rough idea of both what operating systems are supposed to do and what goes on inside them. The rest of the book fills in the details. We return to Sixth-Edition Unix a number of times, using it as an example of simple and often naive ways of doing things — it provides motivation for what's done in recent operating systems.

1.5 CONCLUSIONS

1. Explain how the notion of *interrupt* facilitated the implementation of *time sharing*.
2. A 32-bit processor can address up to 2^{32} bytes of memory. While this was an astronomical amount of memory in 1960, it is well within the means of most home-computer owners today. Virtual memory was invented partly to make it easy for one to run large programs in computers with relatively small amounts of memory. Given that one can readily afford as much real memory as virtual memory, is the concept of virtual memory relevant for 32-bit computers? Explain.
3. How might the world be different today if Microsoft was able to offer Xenix rather than DOS as the operating system for the IBM PC?
4. What might be the distinction between *time sharing* and *preemptive multitasking*?
5. Explain the difference between an interrupt and a trap.
6. If a Unix program creates a new process by calling *fork*, can a bug in the child process, such as a bad array reference, damage the parent process? Explain.
7. Most modern operating systems support multi-threaded processes. Consider Figure 1.2, which shows the Sixth-Edition-Unix address space. What would have to change in this figure for it to show an address space of a multi-threaded process?
8. Suppose, in a program that is to run on Unix, one wants to append data to the end of the file. Are the techniques given in Section 1.3.5 sufficient for doing this? In other words, using these techniques, can you write a program that is guaranteed always to append data to the current end of a file? Explain. (*Hint*: consider the case of multiple unrelated programs appending data to the same file.)
9. The program fragment shown in Section 1.3.5.3 shows how to print a file backwards. Explain why it is not a good technique for doing so. Present a program that does it better.
10. Section 1.3.5.5 mentions that hard links may not point to directories. Explain why not.
11. Symbolic links may point to directories. However, a certain restriction must be placed on their use so as to permit this. What is this restriction?
12. Most shell commands (such as *ls*, *ps*, *rm*, *cp*, etc.) are implemented by creating separate processes to run them. However, this is not the case with the *cd* (change directory) command. Explain why not.
13. Explain how the shell's *pwd* command, which prints the full path name of the current directory, might be implemented.

1.6 EXERCISES

14. Windows lets one access files in a directory even though one is allowed no form of access to the directory's parent. Explain why this is not allowed in Unix. (Windows, as described in Chapter 8, has a different form of access control for which this makes sense.)
15. There are certain public directories, such as /tmp, in which everyone is allowed to create a file. Can one specify using the access-permission bits of Sixth-Edition Unix that only the owner of a file in such a directory can unlink it? Explain and either show how it can be done or suggest a simple modification to Sixth-Edition Unix that would make this possible.

1.7

REFERENCES

- Bach, M. J. (1986). *The Design of the UNIX Operating System*, Prentice Hall.
- Corbató, F. J., M. M. Daggett, R. C. Daley (1962). An experimental Time-Sharing System. *Spring Joint Computer Conference*.
- Corbató, F. J., J. H. Saltzer, C. T. Clingen (1972). Multics — The First Seven Years. *Spring Joint Computer Conference*.
- Dennis, J. B. and E. C. Van Horn (1966). Programming Semantics for Multiprogrammed Computations. *Communications of the ACM* 9(3): 143–155.
- Dijkstra, E. W. (1959). *Communication with an Automatic Computer*, University of Amsterdam.
- Kilburn, T., D. B. G. Edwards, M. J. Lanigan, F. H. Sumner (1962). One-level Storage System. *IRE Transactions, EC-11* 2: 223–235.
- Kildall, G. A. (1981). CP/M: A Family of 8-and 16-Bit Operating Systems. *Byte Magazine*, June 1981.
- Killian, T. J. (1984). Processes as Files. *USENIX Conference Proceedings*, Salt Lake City, UT, USENIX Association: 203–207.
- Lee, J. A. N. (1992). Claims to the Term “Time-Sharing.” *Annals of the History of Computing* 14(1): 16–17.
- McCarthy, J. (1983). Reminiscences on the History of Time Sharing. <http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html>.
- McKusick, M. K., K. Bostic, M. J. Karels, J. S. Quarterman (1996). *The Design and Implementation of the 4.4 BSD Operating System*, Addison Wesley.
- Pike, R., D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, P. Winterbottom (1995). Plan 9 from Bell Labs. *Computing Systems* 8(3): 221–254.
- Rashid, R., R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, R. Sanzi (1989). Mach: A Foundation for Open Systems. *Proceedings of the Second Workshop on Workstation Operating Systems*.
- Russinovich, M. E. and D. A. Solomon (2005). *Microsoft Windows Internals*, Microsoft Press.
- Ryckman, G. F. (1983). The IBM 701 Computer at the General Motors Research Laboratories. *Annals of the History of Computing* 5(2): 210–212.
- Smotherman, M. (2008). Interrupts. <http://www.cs.clemson.edu/~mark/interrupts.html>.
- Strachey, C. (1959). Time Sharing in Large Fast Computers. *Proceedings of International Conference on Information Processing*, UNESCO: 336–341.
- Tanenbaum, A. S. (1987). *Operating Systems: Design and Implementation*, Prentice Hall.
- Wikipedia. (2009). IBM 7030 Stretch. *Wikipedia*, http://en.wikipedia.org/wiki/IBM_7030.
- Zimmermann, H., M. Guillemont, G. Morisett, J.-S. Banino (1984). Chorus: A Communication and Processing Architecture for Distributed Systems, INRIA, Rapports de Recherche 328.

2.1	Why Threads?	2.2.2	Threads and C++
2.2	Programming with Threads	2.2.3	Synchronization
2.2.1	Thread Creation and Termination	2.2.3.1	Mutual Exclusion
2.2.1.1	Creating POSIX Threads	2.2.3.2	Beyond Mutual Exclusion
2.2.1.2	Creating Win-32 Threads	2.2.4	Thread Safety
2.2.1.3	Handling Multiple Arguments	2.2.5	Deviations
2.2.1.4	Thread Termination	2.2.5.1	Unix Signals
2.2.1.5	Attributes and Stack Size	2.2.5.2	POSIX Cancellation
2.2.1.6	Example	2.3	Conclusions
		2.4	Exercises
		2.5	References

In computer systems many things must go on at the same time; that is, they must be **concurrent**. Even in systems with just one processor, execution is generally multiplexed, providing at least the illusion that many things are happening at once. At any particular moment there may be a number of running programs, a disk drive that has completed an operation and requires service, packets that have just arrived from the network and also require service, characters that have been typed at the keyboard, etc. The operating system must divide processor time among the programs and arrange so that they all make progress in their execution. And while all this is going on, it must also handle all the other input/output activities and other events requiring attention as well.

This chapter covers multithreaded programming. The discussion here not only goes through the basics of using concurrency in user-level programs, but also introduces a number of concepts that are important in the operating system. We start by motivating multithreaded programming through the use of some simple, high-level examples. We introduce POSIX¹ threads and Win-32 threads, standards for multithreaded programming in C and C++ on Unix and Windows systems, respectively. As part of our discussion, we introduce synchronization, cancellation, and other threads issues.

Though this is a text on operating systems, we cover multithreaded programming from an application programmer's perspective. This is primarily to facilitate programming exercises — it is much easier to do such exercises as normal application code than as code within an operating system. Though the facilities available for multithreaded programming might be different within an operating system than on top of an operating system, the essential concepts are the same.

¹ POSIX stands for *portable operating system interface* and is a registered trademark of the Institute of Electrical and Electronics Engineers (IEEE).

2.1 WHY THREADS?

We are accustomed to writing single-threaded programs and to having multiple single-threaded programs running on our computers. Why do we want multiple threads running in the same program? Putting it only a bit over-dramatically, programming with multiple threads is a powerful paradigm. It is tempting to say “new paradigm,” but the concept has been around since at least the 1960s — though only since the 1990s has it received serious vendor support.

So, what is so special about this paradigm? Programming with threads is a natural way both to handle and to implement concurrency. As we will see, concurrency comes up in numerous situations. A common misconception is that it is useful only on multiprocessors. Threads do let us exploit the features of a multiprocessor, but they are equally useful on uniprocessors. In many instances a multithreaded solution to a problem is simpler to write, simpler to understand, and simpler to debug than a single-threaded solution to the same problem.

To illustrate this point, let’s look at some code extracted from a program *rlogind* (for “remote login daemon”), in common use on Unix systems. This program is a major component of support for remote access.² Figure 2.1 shows a rough sketch of how remote access works. The idea here is that as you sit at the client machine on the left you have a remote-login session on the server machine on the right. You are running applications on the server that get their keyboard input from the client and send their display output to the client.

To make this work, each character you type is transferred over the communication line to the server; it is read there by the *rlogind* program, which then simply writes the characters out to a special facility called a “pseudoterminal” that makes the characters appear as if they were typed at the server’s keyboard. (We discuss pseudoterminals in detail in Section 4.1.2.1.) Similarly, characters output from applications are written to the pseudo-terminal; they are read from there by the *rlogind* program, which then writes them to the communication line. These characters are then displayed on the client.

Thus the job of the *rlogind* program is simply to process two streams of characters. It reads characters from one stream from the communication line coming from the client and writes them via the pseudoterminal to the applications running on the server. It reads characters from the other stream from the pseudoterminal and writes them to the communication line going to the client.

The C code actually used for this is:

```
void rlogind(int r_in, int r_out, int l_in, int l_out) {
    fd_set in = 0, out;
    int want_l_write = 0, want_r_write = 0;
    int want_l_read = 1, want_r_read = 1;
```

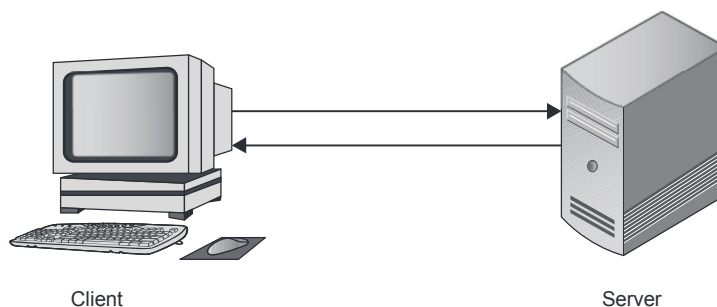


FIGURE 2.1 Remote login.

²It is, however, obsolete, not being secure. We discuss why in Chapter 8.

```

int eof = 0, tsize, fsize, wret;
char fbuf[BSIZE], tbuf[BSIZE];

fcntl(r_in, F_SETFL, O_NONBLOCK);
fcntl(r_out, F_SETFL, O_NONBLOCK);
fcntl(l_in, F_SETFL, O_NONBLOCK);
fcntl(l_out, F_SETFL, O_NONBLOCK);

while(!eof) {
    FD_ZERO(&in);
    FD_ZERO(&out);
    if (want_l_read)
        FD_SET(l_in, &in);
    if (want_r_read)
        FD_SET(r_in, &in);
    if (want_l_write)
        FD_SET(l_out, &out);
    if (want_r_write)
        FD_SET(r_out, &out);

    select(MAXFD, &in, &out, 0, 0);
    if (FD_ISSET(l_in, &in)) {
        if ((tsize = read(l_in, tbuf, BSIZE)) > 0) {
            want_l_read = 0;
            want_r_write = 1;
        } else
            eof = 1;
    }
    if (FD_ISSET(r_in, &in)) {
        if ((fsize = read(r_in, fbuf, BSIZE)) > 0) {
            want_r_read = 0;
            want_l_write = 1;
        } else
            eof = 1;
    }
    if (FD_ISSET(l_out, &out)) {
        if ((wret = write(l_out, fbuf, fsize)) == fsize) {
            want_r_read = 1;
            want_l_write = 0;
        } else if (wret >= 0)
            tsize -= wret;
        else
            eof = 1;
    }
}

```

```

    if (FD_ISSET(r_out, &out)) {
        if ((wret = write(r_out, tbuf, tsize)) == tsize) {
            want_l_read = 1;
            want_r_write = 0;
        } else if (wret >= 0)
            tsize -= wret;
        else
            eof = 1;
    }
}
}

```

It is not immediately apparent what this code does (that's the point!): after some scrutiny, you discover that it reads characters typed by the remote user (arriving on the communication line via file descriptor *r_in*), outputs these characters to local applications (via the pseudoterminal on file descriptor *l_out*), reads characters output by local applications (arriving from the pseudoterminal via file descriptor *l_in*), and sends them to the remote user (on the communication line via file descriptor *r_out*). To ensure that it never blocks indefinitely waiting for I/O on any of the four file descriptors, it makes somewhat complicated use of nonblocking I/O and the *select* system call.

Though this program is conceptually straightforward, the code definitely isn't. Look what happens when we rewrite this single-threaded program as a two-threaded program:

```

void incoming(int r_in, int l_out) {
    int eof = 0;
    char buf[BSIZE];
    int size;

    while (!eof) {
        size = read(r_in, buf, BSIZE);
        if (size <= 0)
            eof = 1;
        if (write(l_out, buf, size) <= 0)
            eof = 1;
    }
}

void outgoing(int l_in, int r_out) {
    int eof = 0;
    char buf[BSIZE];
    int size;

    while (!eof) {
        size = read(l_in, buf, BSIZE);
        if (size <= 0)
            eof = 1;
        if (write(r_out, buf, size) <= 0)
            eof = 1;
    }
}

```

One thread, running the *incoming* procedure, simply reads from *r_in* (the communication line) and writes to *l_out* (the pseudoterminal); the other, running the *outgoing* procedure, reads from *l_in* (the pseudoterminal) and writes to *r_out* (the communication line). This solution and the previous one are equivalent, but the two-threaded implementation is much easier to understand.

Figure 2.2 shows another server application, a single-threaded database server handling multiple clients. The issue here is: how does the single-threaded server handle multiple requests? The easiest approach is, of course, one at a time: it deals completely with the first request, then the second, then the third, and so on. But what if some of the requests take a long time to handle,

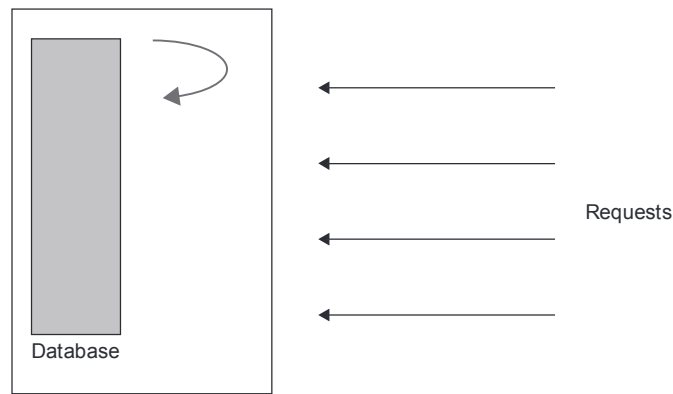


FIGURE 2.2 Single-threaded database server.

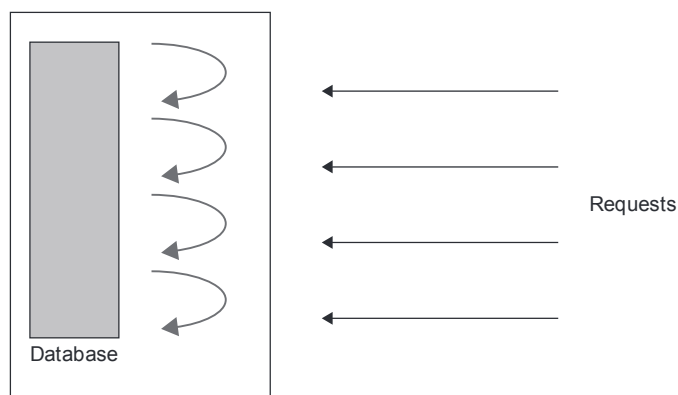


FIGURE 2.3 Multithreaded database server.

while others can be dealt with quickly? What we'd like is to handle these requests concurrently. We can do this by building a timer into the application: our thread would work on the first request for, say, a millisecond, then spend a millisecond on the next request, and so on. Once each pending request receives a millisecond of service, the thread goes back to the first and gives each another millisecond, and so forth. Thus long requests are eventually handled completely, but short requests are dealt with relatively quickly.

It wouldn't be any fun, though, to write the code for this: not only would you have to produce a correct database implementation, but you'd also have to implement the timing and multiplexing correctly. So, instead, let's look at a multithreaded implementation — see Figure 2.3. In this new version of the server, each request is handled by a separate thread. Thus any concurrency in handling the requests is provided by the threads implementation. Of course, someone has to write the code to provide this concurrency, but at least the writer of the threads package to do this didn't simultaneously have to write a correct database implementation. The resulting multithreaded database implementation should be just as efficient as the single-threaded one we did (or considered doing) ourselves. Furthermore, if we have a multiprocessor, our threads can take advantage of it without any additional work by the programmer.

2.2 PROGRAMMING WITH THREADS

Despite the advantages of programming with threads, only relatively recently have standard APIs for multithreaded programming been developed. The most important of these APIs in the Unix world is the one developed by a group originally called POSIX 1003.4a. This multi-year effort resulted in 1995 in an approved standard, now known by the number 1003.1c.

Microsoft produced as part of its Win-32 interface a threads package whose interface has little in common with that of POSIX. Moreover, there are significant differences between the Microsoft and POSIX approaches — some of the constructs of one cannot be easily implemented in terms of the constructs of the other, and vice versa. Despite these incompatibilities, both approaches are useful for multithreaded programming.

2.2.1 THREAD CREATION AND TERMINATION

Creating a thread should be a pretty straightforward operation: in response to some sort of directive, a new thread is created and proceeds to execute code independently of its creator. There are, of course, a few additional details. We may want to pass parameters to the thread. A stack of some size must be created to be the thread's execution context. Also, we need some mechanism for the thread to terminate and to make a termination value available to others.

2.2.1.1 Creating POSIX Threads

POSIX and Win-32 have similar interfaces for creating a thread. Suppose we wish to create a bunch of threads, each of which will execute code to provide some service. In POSIX, we do this as follows:

```
void start_servers( ) {
    pthread_t thread;
    int i;

    for (i=0; i<nr_of_server_threads; i++)
        pthread_create(
            &thread,          // thread ID
            0,                // default attributes
            server,           // start routine
            argument);        // argument
}

void *server(void *arg) {
    // perform service
    return (0);
}
```

Thus a thread is created by calling *pthread_create*. If it returns successfully (returns 0), a new thread has been created that is now executing independently of the caller. This thread's ID is returned via the first parameter (an output parameter that, in standard C programming style, is a pointer to where the result should be stored). The second parameter is a pointer to an attributes structure that defines various properties of the thread. Usually we can get by with the default properties, which we specify by supplying a null pointer. The third parameter is the address of the routine in which our new thread should start its execution. The last argument is the argument that is actually passed to the first procedure of the thread.

If *pthread_create* fails, it returns a positive value indicating the cause of the failure.

2.2.1.2 Creating Win-32 Threads

An equivalent program written for Windows using the Win-32 interface is:

```
void start_servers( ) {
    HANDLE thread;
    DWORD id;
    int i;

    for (i=0; i<nr_of_server_threads; i++)
        thread = CreateThread(
            0,                      // security attributes
            0,                      // default # of stack pages allocated
            server,                 // start routine
            0,                      // argument
            0,                      // creation flags
            &id);                   // thread ID
    }

    DWORD WINAPI server(void *arg) {
        // perform service
        return(0);
    }
}
```

Calls to *CreateThread* are used rather than *pthread_create*. A handle for the new thread is returned. A handle, as we discussed in Chapter 1, is similar to a Unix file descriptor: it refers to information belonging to the user process but maintained in the operating system. In this case, as we'll see, the handle allows the holder to perform operations on the thread.

An ID is returned via the last (output) argument. It is a means of identifying the thread that gives the holder no ability to control that thread. Thus one process can make a thread ID available to another process so as to identify the thread but not give the second process any control over it.

The first parameter is a pointer to the security attributes to be associated with the thread; we use 0 for this for now and discuss other possibilities later. The next parameter is the number of stack pages (in bytes) to allocate physical resources for (one megabyte of virtual memory is allocated; the parameter indicates how much of this initially has real memory and stack space supporting it); 0 means to use the default. The third parameter is the address of the first routine our thread executes; the next parameter is the argument that's passed to that routine. The next to the last parameter specifies various creation flags; we don't supply any here.

If *CreateThread* fails, indicated by returning a null handle, *GetLastError* can be used to determine the cause of the failure.

2.2.1.3 Handling Multiple Arguments

A problem comes up with both *pthread_create* and *CreateThread* when you want to pass more than one argument to a thread. Suppose you are creating threads for use with our two-threaded implementation of *rlogind* (Section 2.1 above). One might be tempted to use the trick outlined below:

```
typedef struct {
    int first, second;
} two_ints_t;
```

```

void rlogind(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;
    two_ints_t in={r_in, l_out}, out={l_in, r_out};
    pthread_create(&in_thread,
        0,
        incoming,
        &in);
    pthread_create(&out_thread,
        0,
        outgoing,
        &out);
}

```

Here we pack two arguments into a structure, then pass a pointer to it to *pthread_create*. This is an example of something that works in single-threaded programs but can cause disastrous failures in multithreaded programs. The variables *in* and *out* are local variables and thus are allocated on the stack of the thread that called *rlogind*. When this first thread returns from *rlogind*, these variables go out of scope — the stack locations might be used for other things. Thus when *pthread_create* is called, the addresses of *in* and *out* point to useful information. But by the time the threads created by the calls to *pthread_create* reference the data pointed to by their arguments (*in* and *out*), this data might no longer exist, since the first thread is no longer in their scope. Thus our approach works only if we can be certain that first thread does not leave the scope of the arguments while they are in use.

Is there a safe way to pass multiple arguments to a thread that works in all cases? Ideally, we'd like to copy all of a thread's arguments onto its stack. But since neither *pthread_create* nor *CreateThread* provides a means for doing this for more than one argument, we need some other technique. (Other threads packages, for example (Doeppner 1987), did provide a way to put multiple arguments on a thread's stack.) Whatever approach we use, it must involve passing a pointer or some sort of pointer-sized identifier to the thread, which then uses this identifier to refer to the actual arguments (which must reside in storage that is available while the thread is executing).

One approach might be to use static or global storage for the arguments, so that there's not a problem with them going out of scope. While this would work in some cases, suppose that in our example multiple threads are calling *rlogind* concurrently. All would use the same locations for storing the arguments to *pthread_create*, and the result would be chaos.

We might allocate storage dynamically for the arguments, using *malloc* in C or *new* in C++. This might seem to solve our problems, but who frees the storage, and when? The creating thread can do so safely only if the created thread is certain not to access the arguments at any point in the future. We can't expect the created thread to free the storage unless its arguments are always in dynamically allocated storage.

In summary, we have four approaches for passing multiple arguments to a thread:

1. copy all arguments to the thread's stack: this always works, but isn't supported in either POSIX or Win-32
2. pass a pointer to local storage containing the arguments: this works only if we are certain this storage doesn't go out of scope until the thread is finished with it
3. pass a pointer to static or global storage containing the arguments: this works only if only one thread at a time is using the storage

4. pass a pointer to dynamically allocated storage containing the arguments: this works only if we can free the storage when, and only when, the thread is finished with it (Note that this is not a problem in languages with automatic garbage collection.)

None of these approaches is suitable in all cases, so we have to figure out which is appropriate for a given situation.

2.2.1.4 Thread Termination

Terminating threads has its share of subtle issues as well. Our threads return values: which threads receive these values and how do they do it? Clearly a thread that expects to receive another's return value must wait until that thread produces it, and this happens only when the other thread terminates. Thus we need a way for one thread to wait until another terminates.³

Though a technique for one thread to wait for any other thread to terminate might seem useful, that threads return values makes it important to be particular about which thread (or threads) one is waiting for. Thus it's important to identify a thread uniquely. Such identification can be a bit tricky if identifiers can be reused.

POSIX provides a rather straightforward construct for waiting for a thread to terminate and retrieving its return value: *pthread_join*. We illustrate its use in a continuation of our example of passing multiple arguments to a thread:

```
void rlogind(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;
    two_ints_t in={r_in, l_out}, out={l_in, r_out};

    pthread_create(&in_thread, 0, incoming, &in);
    pthread_create(&out_thread, 0, outgoing, &out);

    pthread_join(in_thread, 0);
    pthread_join(out_thread, 0);
}
```

Here our thread uses *pthread_join* to insure that the threads it creates terminate before it leaves the scope of their arguments. The first argument to *pthread_join* indicates which thread to wait for and the second argument, if nonzero, indicates where the return value should go.

How exactly does a thread terminate? Or, more precisely, what does a thread do to cause its own termination? (In Section 2.2.5.2 we discuss how one thread can request that another terminate, but for the time being we are concerned only with self-termination.) One way for a thread to terminate is simply to return from its first procedure, returning a value of type *void **. An alternative approach is that a thread call *pthread_exit*, supplying a *void ** argument that becomes the thread's return value.⁴ The following skeletal code illustrates the two approaches:

```
void CreatorProc( ) {
    pthread_t createe;
    void *result;
```

³ An alternate approach might be for a thread to check whether another has terminated and continue on regardless, rather than wait for termination. Though this might occasionally be handy, it's not considered useful enough to warrant inclusion in POSIX threads, though it can be done in Win-32 threads.

⁴ Note that, although returning from the first procedure and calling *pthread_exit* are equivalent in C, they aren't in C++. In particular, calling *pthread_exit* in a C++ program terminates the thread but does not invoke the destructors of any active local objects.

```

    pthread_create(&createe, 0, CreateeProc, 0);
    ...
    pthread_join(create, &result);
    ...
}

void *CreateeProc(void *arg) {
    ...
    if (should_terminate_now)
        pthread_exit((void *)1);
    ...
    return((void *)2);
}

```

There is a big difference between *pthread_exit* and *exit*. The former terminates just the calling thread. The latter terminates the entire process, including all threads running in it. Note that if a thread returns from a program's main procedure, it calls *exit* (it actually returns to code that then calls *exit*). So, consider the following code:

```

int main(int argc, char *argv[ ]) {
    int i;
    pthread_t threads[10];

    for (i=0; i< 10; i++)
        pthread_create(&threads[i], 0, start_proc, (void *)i);

    return(0);
}

```

What happens here is that ten threads are created but the creating thread immediately returns from *main*, calls *exit*, and thus terminates the entire process. On a uniprocessor system, none of the ten child threads are likely to get a chance to run at all.

To create a bunch of threads and then terminate innocuously, one might do something like:

```

int main(int argc, char *argv[ ]) {
    int i;
    pthread_t threads[10];

    for (i=0; i< 10; i++)
        pthread_create(&threads[i], 0, start_proc, (void *)i);

    pthread_exit(0);
    return(0);
}

```

Here the first thread again creates ten threads, but then quietly goes away (the call to *return* is required even though it won't be executed, since *main* is defined as returning an *int*). The process terminates once all its component threads have terminated (or if one of them calls *exit*).

Let's look at thread termination and *pthread_join* a bit more closely. It's tempting to treat them as analogs of process termination and *wait*, as discussed in Chapter 1. However, *wait* can be used only to wait for the termination of a child process, while *pthread_join* can be used to wait for the termination of any thread in the process. But, as with process termination, when a thread terminates some amount of information must be left behind: its thread ID and its return value. Thus terminating threads go into a “zombie” state, just as terminating processes do. Once some other thread calls *pthread_join* on the terminated thread, then all record of it may be released and the thread ID may be reused.⁵

What happens if two threads call *pthread_join* on the same target thread? In light of our discussion above, we see that this could result in a problem. After the first thread calls *pthread_join*, the target thread's ID might be reused. Thus when the second thread calls *pthread_join*, it could well be waiting for the termination of a new thread. Since this is probably not what was desired, it's considered unsafe to make more than one call to *pthread_join* on any particular target thread.

The Win-32 approach to waiting for a thread's termination and getting its return value adds one more step. Like a POSIX thread, a Win-32 thread terminates either by returning from its first procedure or by calling an exit routine, in this case *ExitThread*. One thread can wait for another to terminate by calling either *WaitForSingleObject* or *WaitForMultipleObjects*. These routines are more general than their POSIX equivalents in that they let us wait not just for threads to terminate, but for any “kernel object” to change its state in some well defined fashion (we discuss this further in Section 2.2.3.2). However, unlike *pthread_join*, *WaitForSingleObject* and *WaitForMultipleObjects* don't affect the zombie status of the thread or yield its return value. The latter is done via a call to *GetExitCodeThread* and the former by a call to *CloseHandle*.

Thus creating a thread and performing the equivalent of *pthread_join* in Win-32 is done as follows:

```
void proc ( ) {
    HANDLE thread;
    DWORD ID;

    thread = CreateThread(0, 0, thread_proc, (LPVOID)arg, 0, &id);
    WaitForSingleObject(thread, INFINITE);
    CloseHandle(thread);
}
```

The second argument to *WaitForSingleObject* is a timeout value indicating how many milliseconds to wait before giving up and returning with an error. The special value *INFINITE* means to wait indefinitely.

In some cases, where the value produced by a terminating thread is of no use to anyone, it might not be necessary for any other thread to wait for a thread's termination. For example, a thread might be created to service the request of a client and then quietly terminate. It's rather inconvenient to have to call *pthread_join* to remove these threads once they become zombies, so POSIX provides the *pthread_detach* routine: it marks a thread as *detached* so that when the thread terminates, its thread ID is immediately made available for reuse, and its return value is

⁵ Exactly what the thread ID is depends on the implementation. If it's a 64-bit integer that's incremented by one every time a new thread is created, then it's highly unlikely ever to be reused. But if it's the address of a data structure that represents the thread, then reuse is likely.

ignored. Thus, so that no thread is left permanently in the zombie state, for each call to *pthread_create* there should be exactly one call to either *pthread_join* or *pthread_detach*.⁶

Win-32 uses a more general mechanism to handle whether to wait for a thread's termination. We saw above that calls to *CreateThread* return a handle to the newly created thread object. Another handle is also created that is stored as part of the newly created thread. The thread object maintains a reference count of handles referring to itself; given these two handles, this reference count is initially two. When the thread terminates, its handle to its own thread object is closed, reducing the object's handle count by one. The thread object, which contains the thread's exit code, isn't deleted until its handle count drops to zero. So, if you want a thread object to disappear once the thread terminates (that is, if you want the effect of *pthread_detach*), you simply close the handle returned by *CreateThread* after the thread is created.

2.2.1.5 Attributes and Stack Size

So far, when we've created threads using *pthread_create*, we've used the default attributes. These are normally sufficient, but in some cases we might want different attributes. First of all, what are these attributes? They are various properties of the thread itself that depend on the implementation. Rather than specifying every property as a separate argument to *pthread_create*, they are all bundled into the attributes argument, thus allowing new attributes to be added as necessary. Example attributes (or properties) are the size of the thread's stack, whether it is created as a detached thread or is joinable, and its initial scheduling priority.

We've seen that supplying a zero for the attributes argument creates a thread with default attributes. To get something other than this default, you need to create an attributes structure, then specify in it the desired attributes. To set this up, you call *pthread_attr_init* and then, via operations on the structure, specify the desired attributes. The resulting attributes structure can then be used as an argument to the creation of any number of threads.

The attributes structure affects the thread only when it is created. Modifying this structure later has no effect on already created threads, but only on threads created with it subsequently.

Storage may be allocated as a side effect of calling *pthread_attr_init*. To ensure that this storage is freed, call *pthread_attr_destroy* with the attributes structure as argument. Note that if the attributes structure goes out of scope, not all storage associated with it is necessarily released — to release this storage you must call *pthread_attr_destroy*.

One attribute that is supported by all POSIX threads implementations is the size of a thread's stack. This is a rather important attribute, though one that many programmers give little thought to. The stack that's allocated for the sole thread in single-threaded processes is so large that we're highly unlikely to need a larger one. However, if we are creating a lot of threads in a process, then we must pay more attention to stack size.

For example, suppose we are creating 1024 threads, each with an eight-megabyte stack (the default stack size in some versions of POSIX threads on Linux). This would require eight gigabytes of address space. But a 32-bit machine has only four gigabytes of address space! So if we want that many threads, each one must have a smaller stack.

Of course, we can go too far in the other direction: if we don't give a thread a large enough stack, serious problems result when the thread exceeds its stack space. This is particularly true if a thread goes beyond its allotted stack and starts clobbering other data structures, such as the stacks of other threads. Though most systems mark the last page of each thread's stack as inaccessible so that attempts to reference it are caught by hardware, a thread can jump over the last page of its stack and start writing on what follows without attracting the hardware's attention — this can

⁶ However, one can specify in the attributes passed to *pthread_create* that the created thread is to be detached, thereby making a call to *pthread_detach* unnecessary. Note that when the process terminates, all traces of all threads are eliminated: no thread stays in the zombie state beyond the life of the process.

happen, for example, if the thread places a call to a procedure with large local variables that aren't immediately referenced. So we must be careful to give each thread a sufficiently large stack (how large is large enough depends on the program and the architecture).

To specify the stack size for a thread, one sets up an attributes structure using *pthread_attr_t* *setstacksize* and supplies it to *pthread_create*:

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);

...

pthread_create(&thread, &thr_attr, startroutine, arg);
```

In this case we've created a thread with a twenty-megabyte stack.

How large is the default stack? This is not specified in POSIX threads. Different implementations use different values, ranging from around 20 KB in Tru64 Unix to one megabyte in Solaris and eight megabytes in some Linux implementations.

Win-32 uses a different approach to stack size: the amount of address space allocated for each thread's stack is a link-time parameter, not a run-time parameter. The "stacksize" argument to *CreateThread* indicates how many pages of primary memory are allocated to hold the stack when the thread starts execution. Additional pages of primary memory are allocated as necessary as the thread runs, up to the maximum stack size. One cannot affect this level of detail using POSIX threads.

2.2.1.6 Example

Here is a simple complete multithreaded program that computes the product of two matrices. Our approach is to create one thread for each row of the product and have these threads compute the necessary inner products. (This isn't a good way to compute the product of two matrices — it's merely an example of a multithreaded program!)

```
#include <stdio.h>
#include <pthread.h>      /* all POSIX threads declarations */
#include <string.h>       /* needed to use strerror below */

#define M      3
#define N      4
#define P      5

int A[M][N];      /* multiplier matrix */
int B[N][P];      /* multiplicand matrix */
int C[M][P];      /* product matrix */

void *matmult(void *);

int main( ) {
    int i, j;
```

```

pthread_t thr[M];
int error;

/* initialize the matrices ... */

...

for (i=0; i<M; i++) {          /* create the worker threads */
    if (error = pthread_create(
        &thr[i],
        0,
        matmult,
        (void *)i)) {
        fprintf(stderr, "pthread_create: %s", strerror(error));
        /* this is how one converts error codes to useful text */
        exit(1);
    }
}

for (i=0; i<M; i++) /* wait for workers to finish */
    pthread_join(thr[i], 0)

/* print the results ... */
...

return(0);
}

void *matmult(void *arg) {
    /* computes all inner products for row arg of matrix C */
    int row = (int)arg;
    int col;
    int i;
    int t;

    for (col=0; col < P; col++) {
        t = 0;
        for (i=0; i<N; i++)
            t += A[row][i] * B[i][col];
        C[row][col] = t;
    }
    return(0);
}

```

Note that we check for errors from *pthread_create*: this is important to do since it can fail (for example, because the operating system has run out of resources for representing a thread).

2.2.2 THREADS AND C++

We've introduced multithreaded programming not as part of our programming language, but as a subroutine library. We'd now like to exploit the object facilities of C++ to try to integrate multithreaded programming with the language. The result, while not a complete success, will let us take advantage of much of C++'s object orientedness to help us write multithreaded programs. We present this material in terms of POSIX threads, but it applies equally well to Win-32.

What we want to do is to treat threads as objects, so that creating a new instance of a thread object has the side effect of creating a thread. This is easy. However, combining the termination of threads with the destruction of the encapsulating object is a bit tricky. Consider the code below:

```
class Server {
public:
    Server(int in, int out): in_(in), out_(out) {
        pthread_create(&thread_, 0, start, (void*)this);
    }
private:
    int in_, out_;
    static void *start(Server*);    // it must be static!
    pthread_t thread_;
};

void *Server::start(Server* me) {
    char buf[BSIZE];
    while(!eof) {
        read(me->in_, buf, sizeof(buf));
        write(me->out_, buf, sizeof(buf));
    }
    return(0);
}
```

We have a class *Server* whose constructor creates a thread that executes a member function. Thus every time we create a new instance of *Server*, we get not only the new instance, but also a thread executing within it. We could, of course, dispense with creating a thread inside *Server*'s constructor; instead, we could simply create a thread explicitly after creating a new instance of *Server* and have this externally created thread call *Server*'s start routine. But by creating the thread in the constructor and having it call a private member function, we are absolutely certain that only one thread is active inside the object: the object and thread executing it are not two different things but the same thing. Outside *Server*, no one needs to know such details as how many threads are running inside of it; this is strictly part of *Server*'s implementation.

There are still a few details to explain and work out. The *start* routine of our thread is declared *static*.⁷ This is necessary because a reference to a nonstatic member function actually requires two pieces of information: a pointer to the function's code and a pointer to the object instance. Both *pthread_create* and the thread it creates are designed to work with the only form of a reference to a function understood by C programs: a pointer to the function's code. So, by declaring *start* to be

⁷This is not sufficient for some C++ compilers, such as Solaris's. For them, the start routine passed to *pthread_create* must be declared as *extern "C"*.

static, we can refer to it simply with a pointer to its code. However, we still must refer to the object instance inside *start*; thus we pass to the thread as its only argument the pointer to the instance.

The object's constructor can have any number of arguments — two in this case. This gives us convenient syntax for supplying more than one argument to a thread, though we still have the storage-management issues discussed earlier: we need to make certain that the lifetime of the storage allocated to hold these arguments is the same as the lifetime of the thread. More generally, we must insure that when the thread terminates, the object is deleted, and vice versa. As we saw when we were dealing strictly with C and trying to handle the storage occupied by threads' initial arguments, there is no one way of doing this.

Nevertheless, let's try to extend our definition of *Server* to handle termination and deletion. Our first attempt is:

```
class Server {
public:
    Server(int in, int out): in_(in), out_(out) {
        pthread_create(&thread_, 0, start, (void*)this);
        pthread_detach(thread_);
    }
    ~server( ) { }
private:
    int in, out;
    static void *start(Server*);
    pthread_t thread_;
};

void *start(Server* me) {
    ...
    delete me;
    return(0);
}
```

We immediately detach the thread after creating it, so there's no need for a call to *pthread_join*. Then just before the thread terminates, we delete the object, thus insuring that the object is deleted when the thread terminates. (Of course, we can do this only because *start* is static.)

The above approach makes sense in this example, but wouldn't work if some other thread needs to be able to wait until this one terminates, perhaps in order to receive a return value. Also, since the object is explicitly deleted by its thread, its instances cannot be local, but must be explicitly created with *new* — if they were local, then when the creating thread leaves their scope, they would be automatically deleted, even though they're also explicitly deleted.

To get around this second problem, let's turn things around a bit and put a call to *pthread_join* in the destructor:

```
class Server {
public:
    Server(int in, int out): in_(in), out_(out) {
        pthread_create(&thread_, 0, start, (void*)this);
    }
    ~server( ) {
```



```

        pthread_join(thread_, 0);
    }
private:
    ...
};

```

Now, when any thread deletes the object, either explicitly or by going out of its scope, it invokes the destructor, which waits until the object's thread terminates.

However, since destructors cannot return values, the object's thread has no way to make a return value available to others. To get around this problem, we might do something like:

```

class Server {
public:
    Server(int in, int out): in_(in), out_(out) {
        pthread_create(&thread_, 0, start, (void*)this);
    }
    int join( ) {
        int ret;
        pthread_join(&thread_, &ret);
        return(ret);
    }
private:
    static void *start(Server*);
    ...
};

```

We've encapsulated *pthread_join* in a new member function that returns a value. However, now the programmer must call *join* and then delete the object. This makes us a bit too dependent on the object's internal details (and can cause problems if the user of the object doesn't use it correctly).

We might try to avoid this problem by:

```

class Server {
public:
    Server(int in, int out): in_(in), out_(out) {
        pthread_create(&thread_, 0, start, (void*)this);
    }
    static int join(Server *t) {
        int ret;
        pthread_join(&t->thread_, &ret);
        delete t;
        return(ret);
    }
private:
    static void *start(Server*);
    ...
};

```

We've made *join* static so that we can delete the object from within it. But now we're back to the original problem: that object instances must not be local.

The moral is that we must design the termination technique to suit the problem.

We have one final issue: subclassing thread objects. It would be convenient to design a generic thread class of which we can build subclasses. For example, consider:

```
class BaseThread {
public:
    BaseThread(void *start(void *)) {
        pthread_create(&thread, 0, start, (void*)this);
    }
private:
    pthread_t thread;
};

class DerivedThread: public BaseThread {
    static void *start(void *arg) {
        cout << (DerivedThread*)arg->a;
        return(0);
    }
public:
    DerivedThread(int z):
        a(z), BaseThread((void*)(*) (void*))start() { }

    int a;
};
```

What we have here is a base class, *BaseThread*, whose constructor creates a thread that calls as its first function the argument passed to the constructor. We then declare another class, *DerivedThread*, that's a subclass of *BaseThread*. *DerivedThread*'s constructor takes one argument, which is saved as an instance variable. The constructor also invokes *BaseThread*'s constructor, passing it the start routine that's part of *DerivedThread*.

Note that, according to the rules of C++, base-class constructors are called before derived-class constructors. So, despite the order of initialization given in *DerivedThread*'s constructor, *BaseThread*'s constructor is called before *a* is initialized to *z*. Thus, there's a potential race condition: suppose we create an object of type *DerivedThread*. *BaseThread*'s constructor first creates a thread, then *DerivedThread*'s constructor initializes *a*. There is a good chance that the new thread will refer to *a* before *a* is initialized.

To get around this problem, we'd really like to make certain that the thread created in the base-class constructor doesn't run until the constructors of all subclasses complete. Unfortunately, there's no convenient means for arranging this in C++.

2.2.3 SYNCHRONIZATION

Our next concern is the coordination of threads as they access common data structures. We examine this issue initially from the point of view of POSIX and Win-32 programmers, and later on from that of the operating-system developer. We start with the relatively simple concern of mutually exclusive access to data structures, then look at more complex synchronization issues.



FIGURE 2.4 World War I fighter aircraft. (Copyright © iStockphoto.)

2.2.3.1 Mutual Exclusion

The mutual-exclusion problem involves making certain that two things don't happen at once. A dramatic example arose in the fighter aircraft of World War I, as illustrated in Figure 2.4. Due to a number of constraints (e.g., machine guns tended to jam frequently and thus had to be near people who could unjam them), machine guns were mounted directly in front of the pilot. However, blindly shooting a machine gun through a whirling propeller was not a good idea. At the beginning of the war, pilots, being gentlemen, politely refrained from attacking fellow pilots. A bit later in the war, however, the Germans developed the tactic of gaining altitude on an opponent, diving at him, turning off the engine, then firing — without hitting the now-stationary propeller. Today this would be called *coarse-grained synchronization*. Later, the Germans developed technology that synchronized the firing of the gun with the whirling of the propeller, so that shots were fired only when the propeller blades would not be in the way. This could well be the first example of a mutual-exclusion mechanism providing *fine-grained synchronization*!

For a more computer-oriented example of the need for mutual exclusion, consider two threads, each performing the following operation:

```
x = x+1;
```

If the two threads perform the operation at the same time, what's the final value of x if its initial value was zero? We'd very much like the answer to be 2. However, consider the assembly-language version of the statement. It probably looks something like this:

```
ld    r1,x
add   r1,1
st    r1,x
```

Thus to add 1 to the variable x , the machine must first load the current contents of location x into a register, add 1 to the contents of the register, then store those contents back into the location containing x . If both threads do this at more or less the same time, the final result stored in x is likely to be 1!

For the concurrent execution of the two assignment statements to behave as we want it to, we somehow have to insure that the effect of executing the three assembler instructions is atomic, i.e., that all three instructions take place at once without interference.

We're not going to show how to solve this problem right away. Instead, we introduce functionality from POSIX threads and Win-32 and show how we can use it to solve the problem. In Chapter 5 we show how this functionality is implemented.

POSIX threads defines a new data type called a *mutex*, which stands for mutual exclusion. A mutex is used to insure either that only one thread is executing a particular piece of code at once (code locking) or that only one thread is accessing a particular data structure at once (data locking). A mutex belongs either to a particular thread or to no thread (i.e., it is either locked or unlocked). A thread may lock a mutex by calling *pthread_mutex_lock*. If no other thread has the mutex locked, then the calling thread obtains the lock on the mutex and returns. Otherwise it waits until no other thread has the mutex, and finally returns with the mutex locked. There may, of course, be multiple threads waiting for the mutex to be unlocked. Only one thread can lock the mutex at a time; there is no specified order for which thread gets the mutex next, though the ordering is assumed to be at least somewhat fair.

To unlock a mutex, a thread calls *pthread_mutex_unlock*. It is considered incorrect to unlock a mutex that is not held by the caller (i.e., to unlock someone else's mutex). However, checking for this is costly, so most implementations, if they check at all, do so only when certain degrees of debugging are turned on.

Like any other data structure, mutexes must be initialized. This can be done via a call to *pthread_mutex_init* or can be done statically by assigning *PTHREAD_MUTEX_INITIALIZER* to a mutex. The initial state of such initialized mutexes is unlocked. Of course, a mutex should be initialized only once! (That is, make certain that, for each mutex, no more than one thread calls *pthread_mutex_init*.) If a mutex is dynamically initialized, a call should be made to *pthread_mutex_destroy* when it is no longer needed.

Using mutexes, we can solve the problem of atomically adding 1 to a variable.

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
                // shared by both threads
int x;          // ditto

pthread_mutex_lock(&m);

x = x+1;

pthread_mutex_unlock(&m);
```

Using a mutex to provide mutually exclusive access to a single data structure at a time is easy. Things get a bit more involved if we need to arrange for such access to multiple data structures at once. Consider the following example in which one thread is executing *proc1* and another is executing *proc2*:

```
void proc1( ) {
    pthread_mutex_lock(&m1);
    /* use object 1 */
    pthread_mutex_lock(&m2);
    /* use objects 1 and 2 */
    pthread_mutex_unlock(&m2);
    pthread_mutex_unlock(&m1);
}

void proc2( ) {
    pthread_mutex_lock(&m2);
    /* use object 2 */
    pthread_mutex_lock(&m1);
    /* use objects 1 and 2 */
    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);
}
```

The threads are using two mutexes to control access to two different objects. Thread 1 first takes mutex 1, then, while still holding mutex 1, obtains mutex 2. Thread 2 first takes mutex 2, then, while still holding mutex 2, obtains mutex 1.

However, things do not always work out as planned. Say that thread 1 obtains mutex 1 and, at about the same time, thread 2 obtains mutex 2; then if thread 1 attempts to take mutex 2 and thread 2 attempts to take mutex 1, we're stuck: we have a *deadlock*.

Deadlock is an important concept, so let's make sure we understand it. Let's say we have a set of threads and a set of containers. Each container contains some number of items. A thread can take items from any of the containers. Threads may put the items back into the containers (and will, unless there is a deadlock). Furthermore, threads may wait until containers have the requisite number of items before taking items out. In our example, containers are mutexes and each contains at most one item. Locking a mutex corresponds to waiting until a container has an item, then taking that item — removing it from the container. Unlocking a mutex corresponds to putting the item back in the container. This is very simple, but things can get much more complicated. For example, Windows (usefully) allows threads to wait for any or all of a set of containers to have items before taking one or more of the items (see the discussion of *WaitForMultipleObjects* towards the end of Section 2.2.3.2 below).

For deadlock to be possible, the following must be true:

1. It must be possible for a thread to hold items it has removed from various containers while waiting for items to become available in other containers.
2. A thread cannot be forced to yield the items it is holding.
3. Each container has a finite capacity.
4. A *circular wait* can exist. We say that thread A is *waiting on* thread B if B is holding items from containers from which A is waiting for items. Consider all threads Z for which there is some sequence of threads B, C, D, . . . , Y, such that A is waiting on B, B is waiting on C, C is waiting on D, . . . , and Y is waiting on Z. The set of such threads Z is known as the *transitive closure* of the relation *waiting on*. If A is contained in this transitive closure, then we say that a *circular wait* exists.

In their full generality, the above conditions are necessary but not sufficient for deadlock to occur. However, in the case of simple operations on mutexes, where threads can wait on only one of them at a time (while holding locks on any number of them), these conditions are sufficient as well.

If any of the four above conditions does not hold, then deadlock cannot happen, even in the most general case. Of course the only nontrivial condition to check for is the last. Algorithms exist to determine whether a circular wait currently exists and to determine whether a particular next move will inevitably result in a circular wait. However, for most purposes, even for simple operations on mutexes, such algorithms are too expensive, since they require time quadratic in the number of threads and mutexes. Let's restrict ourselves to simple operations on mutexes and, rather than trying to solve the problem for arbitrary programs, let's simply write our code in an intelligent fashion so that circular waits, and hence deadlocks, cannot happen.

Let's represent the state of our program as a directed bipartite graph, with one type of node representing threads and the other representing mutexes (*bipartite* simply means that there are two types of nodes, and edges can go only from one type to the other). We draw an edge from a mutex to a thread if the thread has the mutex locked; and we draw an edge from a thread to a mutex if the thread is waiting to lock the mutex. A circular wait and hence deadlock are represented by the graph in Figure 2.5. It should be clear that a cycle exists in such a graph if and only if there is a circular wait. Thus to prevent deadlock, we must make certain that cycles

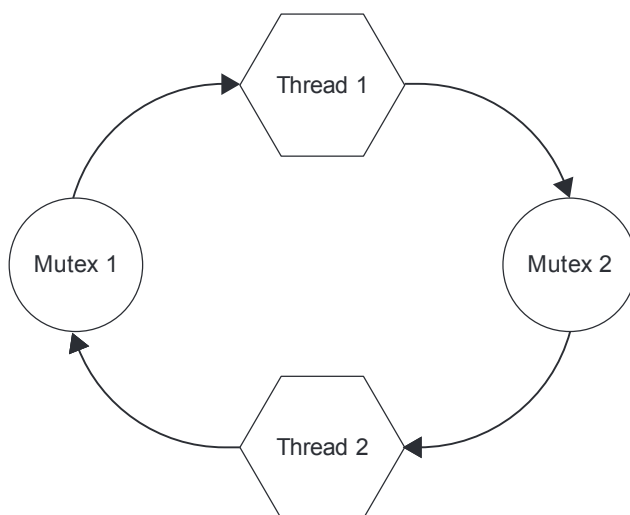


FIGURE 2.5 Thread/mutex graph.

can't possibly exist in the graph. This is easy to do: we simply arrange our mutexes in some order (for example, by assigning unique integers to them) and insist that all threads attempting to lock multiple resources do so in ascending order.

In most applications it is fairly easy to set things up so that all threads lock mutexes in ascending order. However, in some situations this is impossible, often because it is not known which mutex should be locked second until the first one is locked. An approach that often works in such situations is to use conditional lock requests, as in:

```

proc1( ) {
    pthread_mutex_lock(&m1);
    /* use object 1 */
    pthread_mutex_lock(&m2);
    /* use objects 1 and 2 */
    pthread_mutex_unlock(&m2);
    pthread_mutex_unlock(&m1);
}

proc2( ) {
    while (1) {
        pthread_mutex_lock(&m2);
        if (!pthread_mutex_trylock(&m1))
            break;
        pthread_mutex_unlock(&m2);
    }

    /* use objects 1 and 2 */

    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);
}

```

Here thread 1, executing *proc1*, obtains the mutexes in the correct order. Thread 2, executing *proc2*, must for some reason take the mutexes out of order. If it is holding mutex 2, it must be careful about taking mutex 1. So, rather than call *pthread_mutex_lock*, it calls *pthread_mutex_trylock*, which always returns without blocking. If the mutex is available, *pthread_mutex_trylock* locks the mutex and returns 0. If the mutex is not available (that is, if it is locked by another thread), then *pthread_mutex_trylock* returns a nonzero error code (EBUSY). In the example above, if

mutex 1 is not available, this is probably because it is currently held by thread 1. If thread 2 were to block waiting for the mutex, we have an excellent chance for deadlock. So, rather than block, thread 1 not only quits trying for mutex 1 but also unlocks mutex 2 (since thread 1 could well be waiting for it). It then starts all over again, first taking mutex 2, then mutex 1.

Thread 2 thus repeatedly tries to lock both mutexes (in the wrong order) until it can do so without causing any problems. This could, of course, require a fair number of iterations. When this approach is used, the assumption (which must be validated) is that contention for locks is low and thus even two iterations are unlikely to occur. If lock contention is high, another solution is necessary, perhaps one that requires all threads to honor the locking order.

2.2.3.2 Beyond Mutual Exclusion

Though mutual exclusion is the most common form of synchronization, there are numerous situations that it cannot handle. One obvious extension to mutual exclusion is what's known as the *readers-writers problem*: rather than requiring mutual exclusion for all accesses to a data structure, we can relax things a bit and insist on mutual exclusion only if the data structure is being modified. Thus any number of threads (readers) may be just looking at the data structure at once, but any thread intending to modify it must have mutually exclusive access.

Another common (at least in texts such as this) synchronization problem is the *producer-consumer problem* (sometimes called the *bounded-buffer problem*). Here we have a buffer containing a finite number of slots. As shown in Figure 2.6, a producer is a thread that wishes to put an item into the next empty slot of the buffer. A consumer is a thread that wishes to remove an item from the next occupied slot. The synchronization issue for producers is that if all slots in the buffer are occupied, then producer threads must wait until empty slots are available. Similarly, if all slots are empty, consumer threads must wait until occupied slots are available.

The next synchronization problem might seem a bit mundane, but it's particularly important in many operating systems. It doesn't have a common name; here we call it the *event* problem. A number of threads are waiting for a particular event to happen. Once the event has happened, we'd like to release all of the waiting threads. For example, a number of threads might be waiting for a read operation on a disk to complete. Once it's completed, all threads are woken up and can use the data that was just read in.

Semaphores These problems, and many others, were first identified in the 1960s. The person who did much of the early work in identifying and elegantly solving these problems was Edsger Dijkstra. The *semaphore*, a synchronization operation (Dijkstra undated: early 1960s) he described and used in an early system (Dijkstra 1968), has proven so useful that it continues to be used in most modern operating systems.

A semaphore is a nonnegative integer on which there are exactly two operations, called by Dijkstra *P* and *V*. (What *P* and *V* stand for isn't obvious. While Dijkstra was Dutch and based his terminology on Dutch words, the mnemonic significance of *P* and *V* seems to be lost even

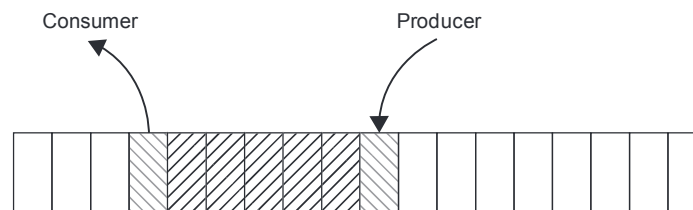


FIGURE 2.6 Producer-consumer problem.

on native Dutch speakers. According to (Dijkstra undated: early 1960s), P stands for *prolagen*, a made-up word derived from *proberen te verlagen*, “try to decrease” in Dutch. V stands for *verhogen*, “increase” in Dutch.) The P operation is somewhat complicated: a thread waits until it finds the value of the semaphore positive, then subtracts one from it. What’s really important is that when the active part of the operation starts, the semaphore’s value is definitely positive, and when this active part finishes, the value is exactly one less than when the operation started. It’s often described as an *atomic* or *indivisible* operation: it has no component parts and takes place as if it were happening instantaneously.

We use the following notation to describe the semantics of P:

```
when (semaphore > 0) [
    semaphore = semaphore - 1;
]
```

This notation means that the operations in square brackets take place only when the expression following “when,” known as the *guard*, is true; the statements in square brackets, known as the *command sequence*, are effectively executed instantaneously: no other operation that might interfere with it can take place while it is executing. We call the entire construct a *guarded command*.

The V operation is simpler: a thread atomically adds one to the value of the semaphore. We write this as

```
[semaphore = semaphore + 1]
```

There is no other means for manipulating the value of the semaphore (other than initializing its value in the first place). Thus if the semaphore’s value is initially one and two threads concurrently execute P and V operations on it, the resulting value of the semaphore is guaranteed to be one. If its value is initially zero and the two threads concurrently execute P and V operations, the P operation must wait until the V operation makes the semaphore’s value positive. Then the P operation can complete, reducing the semaphore’s value to zero.

We can easily implement mutexes using semaphores:

```
semaphore S = 1;
void OneAtATime( ) {
    P(S);
    ...
    /* code executed mutually exclusively */
    ...
    V(S);
}
```

If two threads call the *OneAtATime* routine, the first one to execute the P operation finds S to be one, and so subtracts one from it, making it zero. If the second thread now attempts to execute the P operation, it finds S to be zero and thus must wait. Eventually the first thread performs the V operation, adding one back to S, which enables the second thread to continue. It subtracts one from S and eventually executes the V and adds one back to it. When semaphores are used in such a way that their values are only zero and one, as here, they are known as *binary semaphores*.

When multiple threads attempt to execute a P operation, the effect must be as if they execute it one at a time while the semaphore’s value is positive. If its value is zero, they are queued up and, one at a time, they complete the P operation in response to V operations by other threads.

We can easily generalize our above example to allow up to N threads to execute a block of code (or access a data structure) simultaneously:

```
semaphore S = N;

void NAtATime( ) {
    P(S);
    ...
    /* No more than N threads here at once */
    ...
    V(S);
}
```

Semaphores used this way are known as *counting semaphores*.

A more interesting example of counting semaphores involves using them to solve the producer-consumer problem (see Figure 2.6 above). To keep things simple, let's assume we have one producer and one consumer. We have a buffer with B slots. We use two semaphores: *empty*, representing the number of empty slots, and *occupied*, representing the number of occupied slots. Our solution is:

```
Semaphore empty = B;
Semaphore occupied = 0;
int nextin = 0;
int nextout = 0;

void Produce(char item) {
    P(empty);
    buf[nextin] = item;
    nextin = nextin + 1;
    if (nextin == B)
        nextin = 0;
    V(occupied);
}

char Consume( ) {
    char item;
    P(occupied);
    item = buf[nextout];
    nextout = nextout + 1;
    if (nextout == B)
        nextout = 0;
    V(empty);
    return(item);
}
```

The first P operation of *Produce* causes the thread to wait until there is at least one empty slot in the buffer; at this point the slot is “taken” and *empty*'s value is reduced by one. The producer puts its item in the buffer, then indicates there is one more occupied slot by performing a V operation on *occupied*. Similar actions take place inside of *Consume*.

POSIX provides an implementation of semaphores, but, strangely, it's not part of POSIX threads (POSIX 1003.1c) but part of the POSIX real-time specification (POSIX 1003.1b). This matters only because, depending on the system, you may need to include an additional library when you use semaphores.

The POSIX interface for semaphores is given below.

```
sem_t semaphore;
int err;
```

```

err = sem_init(&semaphore, pshared, init);
err = sem_destroy(&semaphore);
err = sem_wait(&semaphore);           // P operation
err = sem_trywait(&semaphore);        // conditional P operation
err = sem_post(&semaphore);           // V operation

```

Thus semaphores are of type *sem_t*. All operations on them return an integer error code. They must be dynamically initialized using *sem_init* (there is no static initialization such as is provided for mutexes). They take two arguments in addition to the semaphore itself: a flag, *pshared*, indicating whether the semaphore is to be used by threads of just one process (*pshared* = 0) or by threads of multiple processes (*pshared* = 1). We assume the former for now, and discuss the latter later. Once a semaphore is no longer used, *sem_destroy* should be called to free whatever storage was allocated by *sem_init*.

The *sem_wait* operation is the P operation described above. What's new is the *sem_trywait* operation, which is similar to *pthread_mutex_trylock*: if the semaphore's value is positive and thus no waiting is required, it behaves just like *sem_wait* (the value of the semaphore is immediately reduced by one). However, if the semaphore's value is zero, then, rather than waiting, the caller returns immediately with an error code (the value *EAGAIN*).

We discuss the Win-32 version of semaphores in the section on Win-32 Events, below.

POSIX Condition Variables Semaphores are a convenient way to express solutions to a number of synchronization problems, but using them can force amazingly complex solutions for other problems. Thus most operating systems provide additional synchronization constructs. Here we describe POSIX's *condition variables*; later we discuss the *events* of Win-32.

We described the semantics of semaphores using guarded commands. A general implementation of our guarded-command construct is, however, a rather tall order. Somehow we'd have to monitor the values of all variables appearing in the guard (i.e., the expression following the *when*) so as to find out when the guard becomes true. Then we'd have to make certain it remains true when we start executing the command sequence (the code in square brackets that follows), and make certain that this execution is indivisible.

Condition variables give programmers the tools needed to implement guarded commands. A condition variable is a queue of threads waiting for some sort of notification. Threads waiting for a guard to become true join such queues. Threads that do something to change the value of a guard from false to true can then wake up the threads that were waiting.

The following code shows the general approach:

Guarded command

```

when (guard) [
    statement 1;
    ...
    statement n;
]

```

POSIX implementation

```

pthread_mutex_lock(&mutex);
while (!guard)
    pthread_cond_wait(
        &cond_var, &mutex);
statement 1;
...
statement n;
pthread_mutex_unlock(&mutex);

```

```
// code modifying the guard:
...
```

```
pthread_mutex_lock(&mutex);
// code modifying the guard:
...
pthread_cond_broadcast(
    &cond_var);
pthread_mutex_unlock(&mutex);
```

To evaluate the guard safely in the POSIX version, a thread must have mutually exclusive access to all of its components. This is accomplished by having it lock a mutex. If the guard is true, then, with the mutex still locked, it executes the command sequence.

The interesting part is, of course, when the guard is false and thus the thread must wait. In this case, the thread calls *pthread_cond_wait*, which has a complicated effect. What we want to happen is that the thread waits until the guard becomes true. While it's waiting, it's pretty important that it not have the mutex locked (otherwise other threads can't change the value of the guard!). So, *pthread_cond_wait* initially does two things: it unlocks the mutex given as its first argument, and puts the calling thread to sleep, queuing it on the queue represented by the condition variable (of type *pthread_cond_t*) given as its second argument.

Now, assume at least one thread is queued on the condition variable (and thus sleeping). If some other thread causes the guard to become true, we'd like to wake up the waiting threads. This guard-modifying thread first must lock the mutex protecting the guard (we must provide mutually exclusive access for all threads that examine or manipulate the guard's components). Once it makes the guard true, it notifies all threads waiting on the associated condition variable by calling *pthread_cond_broadcast*. Since it's now finished with the guard, it unlocks the mutex.⁸

The threads waiting inside *pthread_cond_wait* are woken up in response to the notification by *pthread_cond_broadcast*, which is intended to inform these threads that the guard, at least a moment ago, evaluated to true. Of course, by the time these threads get a chance to execute, the guard might evaluate to false again. For example, the code in the command sequence executed by the first of the threads to continue execution might do something to negate the guard for the others. So, to guarantee that the guard is true when a thread begins execution of the command sequence, it must reevaluate it, and of course must do so with the mutex locked.

Thus, in response to notifications from *pthread_cond_broadcast* (or from *pthread_cond_signal*, discussed soon), the threads waiting in *pthread_cond_wait* wake up and, still inside *pthread_cond_wait*, immediately call *pthread_mutex_lock* on the mutex they unlocked when they called *pthread_cond_wait* in the first place. As each thread obtains the lock on the mutex, it returns from *pthread_cond_wait*. (We said its effect was complicated!) Now with mutually exclusive access to the components of the guard, the thread can reevaluate it and go back to sleep (by calling *pthread_cond_wait*), if necessary. Thus the notification provided by *pthread_cond_broadcast* must be considered as merely a hint that the guard might now be true. A waiting thread must reevaluate the guard on its own to verify it.

If the first thread released from within *pthread_cond_wait* always makes the guard false from within its command sequence, it seems silly to wake up all the other waiting threads needlessly.

⁸ Some argue that the mutex should be unlocked first and then the call should be made to *pthread_cond_broadcast*, since this would improve concurrency. We prefer the order shown, but both orders work fine.

Thus an alternative to *pthread_cond_broadcast* is *pthread_cond_signal*, which wakes up just the first thread waiting in the condition-variable queue.

The actual POSIX specification of *pthread_cond_wait* allows implementations to take some shortcuts. Since a thread's returning from *pthread_cond_wait* is just an indication that the guard *might* be true, POSIX allows this routine to return spontaneously, i.e., even if no call has been made to either *pthread_cond_signal* or *pthread_cond_broadcast* (though the thread must lock the mutex before returning). This causes no problems other than inefficiency when condition variables are used as described above, but does rule out other potential uses of condition variables, as we discuss below. Note that such spontaneous wakeups are intended to be rare. They don't occur in most current implementations, though there's no guarantee that they won't occur in tomorrow's implementation. See Exercise 6 for some insight on why POSIX allows such shortcuts.

Let's summarize what we've learned about the operations on condition variables:

- *pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *m)* causes the caller to unlock mutex *m*, queue itself on condition variable *cv*, and go to sleep. When the caller wakes up, it calls *pthread_mutex_lock* on mutex *m* and returns from *pthread_cond_wait*. Wakeups occur in response to calls made by other threads to *pthread_cond_signal* and *pthread_cond_broadcast* using the same condition variable *cv*, or for no reason whatsoever.
- *pthread_cond_signal(pthread_cond_t *cv)* causes the first thread queued on condition variable *cv* to be woken up. If there are no such queued threads, nothing happens.
- *pthread_cond_broadcast(pthread_cond_t *cv)* causes all threads queued on condition variable *cv* to be woken up. If there aren't any, nothing happens.

Two more routines are needed:

- *pthread_cond_init(pthread_cond_t *cv, pthread_condattr_t *attr)* dynamically initializes condition variable *cv* according to attributes *attr*. Supplying zero for *attr* causes the default attributes to be used. An option provided by many implementations is the shared use of a condition variable by threads of different processes. Static initialization may also be used (using *PTHREAD_COND_INITIALIZER*), which establishes default attributes.
- *pthread_cond_destroy(pthread_cond_t *cv)* should be called for any dynamically initialized condition variable once it is no longer needed.

The readers-writers problem mentioned earlier is a good example of a conceptually simple synchronization problem that lacks a simple solution using semaphores, but can be solved easily using condition variables. Below is a solution written using guarded commands:

```

void reader( ) {
    when (writers == 0) [
        readers++;
    ]

    // read

    [readers--;]
}

void writer( ) {
    when ((writers == 0) &&
        (readers == 0)) [
        writers++;
    ]

    // write

    [writers--;]
}

```

Here *readers* represents the number of threads that are currently reading and *writers* represents the number of threads that are currently writing. Both are initially zero. Readers must wait until there

are no writers; then they proceed, after indicating that there is one more reader. After they finish reading, they reduce the count of readers by one. Similar actions are taken by writers, but they must wait until there are both no current readers and no current writers.

Guided by our general approach to implementing guarded commands with condition variables, we implement this pseudocode with the following:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t readerQ = PTHREAD_COND_INITIALIZER;
pthread_cond_t writerQ = PTHREAD_COND_INITIALIZER;
int readers = 0;
int writers = 0;

void reader( ) {
    pthread_mutex_lock(&m);
    while (!(writers == 0))
        pthread_cond_wait(
            &readerQ, &m);
    readers++;
    pthread_mutex_unlock(&m);
    // read
    pthread_mutex_lock(&m);
    if (--readers == 0)
        pthread_cond_signal(
            &writerQ);
    pthread_mutex_unlock(&m);
}

void writer( ) {
    pthread_mutex_lock(&m);
    while (!((readers == 0) &&
        (writers == 0)))
        pthread_cond_wait(
            &writerQ, &m);
    writers++;
    pthread_mutex_unlock(&m);
    // write
    pthread_mutex_lock(&m);
    writers--;
    pthread_cond_signal(
        &writerQ);
    pthread_cond_broadcast(
        &readerQ);
    pthread_mutex_unlock(&m);
}
```

Here reader threads wait on the condition variable *readerQ* while *writers* is not zero, indicating the presence of an active writing thread. Writer threads wait on the condition-variable *writerQ* while *readers* and *writers* are both not zero, indicating that either some number of threads are reading or a thread is writing. When a reader thread is finished reading and discovers that it's the last, it wakes up the first waiting writer. When a writer thread is finished writing, it wakes up the first waiting writer and all the waiting readers. Depending on which of these locks the mutex first, either the readers are allowed to proceed or the next writer is allowed to proceed.

To complete our discussion of the readers-writers problem, note that the solution we've presented here might have a problem. Suppose that, whenever the last reader is just about to leave, a new reader arrives. Thus at all times there is at least one reader reading. This is unfortunate for the writers, since they will never get a chance to write.

If reading is a rare event, then such a scenario is highly unlikely. However, often reading is not a rare event but writing is. We'd really like to make certain that our writers get to write quickly, even if this means giving them favored treatment over readers. If the above solution is seen as a "readers-priority" solution, what we'd really like is a "writers-priority" solution.

Devising such a solution is fairly easy. We first solve it with guarded commands:

```

void reader( ) {
    when (writers == 0) [
        readers++;
    ]

    // read

    [readers--;]
}

void writer( ) {
    [writers++;]
    when((active_writers == 0) &&
        (readers == 0)) [
        active_writers++;
    ]

    // write

    [writers--; active_writers--;]
}

```

Here we've changed the meaning of *writers* to represent the number of threads that are either writing or waiting to write. We introduce a new variable, *active_writers*, to represent the number of threads (zero or one) that are currently writing. Reader threads must now wait until there are no threads either writing or waiting to write. Writer threads wait, as before, until no other threads are reading or writing.

Implementing this in POSIX is straightforward:

```

    pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
    pthread_cond_t readerQ = PTHREAD_COND_INITIALIZER;
    pthread_cond_t writerQ = PTHREAD_COND_INITIALIZER;
    int readers = 0;
    int writers = 0;
    int active_writers = 0;

void reader( ) {
    pthread_mutex_lock(&m);
    while (!(writers == 0))
        pthread_cond_wait(
            &readerQ, &m);
    readers++;
    pthread_mutex_unlock(&m);
    // read
    pthread_mutex_lock(&m);
    if (--readers == 0)
        pthread_cond_signal(
            &writerQ);
    pthread_mutex_unlock(&m);
}

void writer( ) {
    pthread_mutex_lock(&m);
    writers++;
    while((!(readers == 0) &&
        (active_writers == 0)))
        pthread_cond_wait(
            &writerQ, &m);
    active_writers++;
    pthread_mutex_unlock(&m);
    // write
    pthread_mutex_lock(&m);
    active_writers--;
    if (--writers == 0)
        pthread_cond_broadcast(
            &readerQ);
    else
        pthread_cond_signal(
            &writerQ);
    pthread_mutex_unlock(&m);
}

```

In this new version, since the variable *writers* indicates whether threads are waiting to write, threads that are leaving the writing code no longer have to notify both waiting readers and the next waiting writer, but do one or the other as appropriate.

The original POSIX threads specification did not include readers-writers locks. However, a more recent version, POSIX 1003.1j, does.

We conclude our discussion of condition variables with an example showing their limitations: the *barrier problem*, a special case of the event problem mentioned above. A barrier is a synchronization construct set up for a specified number of threads, say n . Threads enter the barrier but cannot exit until all n have entered. Once all n threads have entered and they begin to exit, the barrier resets itself so that as threads enter again, they cannot exit until all have entered again, and so forth.

One might be tempted to solve the problem as follows:

```
int count;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;

void barrier( ) {
    pthread_mutex_lock(&m);
    if (++count < n)
        pthread_cond_wait(&BarrierQueue, &m);
    else {
        count = 0;
        pthread_cond_broadcast(&BarrierQueue);
    }
    pthread_mutex_unlock(&m);
}
```

The intent here is that all but the last thread to call *barrier* wait by queuing themselves on the condition variable *BarrierQueue*, and then that the last thread wake up all the others. Thus we're using *pthread_cond_broadcast* to notify the waiting threads that a particular event has happened — all n threads have entered the barrier. But this is not how we're supposed to use condition variables. Returning from *pthread_cond_wait* is merely an indication that the guard we're waiting for might now be true. But since *pthread_cond_wait* might return spontaneously, we can't depend on its return to indicate that what we're interested in is true.

We might try rewriting this code so that threads evaluate a guard each time they return from *pthread_cond_wait*:

```
int count;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;

void barrier( ) {
    pthread_mutex_lock(&m);
    if (++count < n)
        while(count < n)
            pthread_cond_wait(&BarrierQueue, &m);
    else {
        count = 0;
    }
}
```

```

        pthread_cond_broadcast(&BarrierQueue);
    }
    pthread_mutex_unlock(&m);
}

```

This solves the problem of a spontaneous return from *pthread_cond_wait*, but now we have a new problem: the last thread to call *barrier* resets *count* to zero. Now when the waiting threads wake up, they will find the guard false and go back to sleep.

We could try moving “count = 0” someplace else, but there really is no place to put it that makes the code work. So we’re stuck — at least if we insist on using this approach. What we need is either a more appropriate synchronization construct or a guarantee that in this and all future implementations of POSIX threads, *pthread_cond_wait* never returns spontaneously. Neither is likely to happen soon (on POSIX systems).

Certainly one can solve this problem using condition variables, but the solution is far from obvious:

```

int count;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
int generation;

void barrier( ) {
    int my_generation;
    pthread_mutex_lock(&m);
    if (++count < n) {
        my_generation = generation;
        while(my_generation == generation)
            pthread_cond_wait(&BarrierQueue, &m);
    } else {
        count = 0;
        generation++;
        pthread_cond_broadcast(&BarrierQueue);
    }
    pthread_mutex_unlock(&m);
}

```

What we’ve done here is introduce a new guard, independent of *count*, that holds its value long enough for all threads to exit the barrier. Note that *my_generation* is a local variable and thus each thread has its own private copy on its stack. We assume that if *generation* overflows, it becomes zero without causing any exceptions.

Again, the original POSIX threads specification did not include barriers, but POSIX 1003.1j does.

Win-32 Events As might be expected, Win-32’s approach to synchronization is different from POSIX’s, though both provide mutexes and semaphores. We’ve seen that POSIX’s condition

variables are intended not to indicate the occurrence of an event, but to provide notification that some guard or condition might be true. The notified thread must then verify it. The approach taken in Win-32 is simply to provide an indication that some event has happened. This results in a somewhat different way to solve some synchronization problems — in many cases, a more straightforward way.

Win-32 events have to do with state changes in various sorts of kernel objects, such as the thread objects we encountered in Chapter 1. Threads may wait for events to take place in any number (within limits) of objects at once, subject to programmer-specified timeouts. A partial list of the kinds of objects that support waiting for events is given in Table 2.1.

As we saw in Chapter 1, one thread can wait for another to terminate by calling *WaitForSingleObject*:

```
HANDLE NewThread = CreateThread(...);
WaitForSingleObject(NewThread, INFINITE);
```

In general, *WaitForSingleObject* returns either because it has timed out (as specified by its second argument) or because the handle given as its first argument refers to an object in some sort of object-dependent signaled state (terminated, in the case of threads). A thread can wait for a number of threads to terminate by using *WaitForMultipleObjects*:

```
HANDLE NewThreads[n];
int i;
BOOL All;

for (i = 0; i < n; i++)
    NewThreads[i] = CreateThread(...);

WaitForMultipleObjects(n, NewThreads, All, INFINITE);
```

Here our creating thread waits for either all or any one of the threads to terminate, depending on whether *All* is TRUE or FALSE. In the latter case, the call returns a code indicating which of the handles in the array passed as the second argument refers to a terminated thread — the index is, strangely, the return value minus the constant *WAIT_OBJECT_0*. If more than one thread has terminated, repeated calls to *WaitForMultipleObjects* are required to identify

Table 2.1 Kernel objects supporting events.

Type of Object	Events
process	termination
thread	termination
mutex	unlocked
semaphore	value is positive
event	event is “signaled”
waitable timers	timer expiration

them. As with *WaitForSingleObject*, the last argument of *WaitForMultipleObjects* indicates how many milliseconds to wait before timing out, in which case the call returns with the error code *WAIT_TIMEOUT*.

Win-32 mutexes behave like the POSIX versions, for example:

```
HANDLE mutex = CreateMutex(0, FALSE, 0);

void AccessDataStructure( ) {
    WaitForSingleObject(mutex, INFINITE);

    // code executed mutually exclusively

    ReleaseMutex(mutex);
}
```

The call to *CreateMutex* returns the handle of a newly created kernel mutex object. These objects, like other kernel objects, are either in the signaled or unsignaled state; for mutexes, signaled means unlocked and unsignaled means locked. *CreateMutex*'s first argument specifies security attributes; since these are not of concern in this chapter, we set it to zero, meaning use the default. The last argument allows one to give the object a name. This is useful if the object is to be shared by multiple processes, but since again this is not of concern in this chapter, we leave it as zero, giving the object no name. The second parameter indicates whether the mutex is created in the locked state (locked by the creator). We set it to *FALSE*, indicating unlocked.

To lock the mutex, a thread must first wait for it to be unlocked. Calling *WaitForSingleObject* does this, with the side effect of locking it on return. If multiple threads are waiting, they wait their turn, returning one at a time. We could also use *WaitForMultipleObjects*, where the mutex's handle is one of the elements of the handle array. In this case the mutex is locked if the call returns with an indication that the mutex was in the signaled state (meaning it was unlocked, but is now locked). To unlock a mutex, simply call *ReleaseMutex* and thus put the object into the *signaled* state.

Semaphores are supported in a similar fashion, for example

```
HANDLE semaphore = CreateSemaphore(0, 1, 1, 0);

void AccessItWithASemaphore( ) {
    WaitForSingleObject(semaphore, INFINITE);

    // code executed mutually exclusively

    ReleaseSemaphore(semaphore, 1, 0);
}
```

The call to *CreateSemaphore* returns the handle of a newly created kernel semaphore object that is considered to be in the signaled state if its value is positive and unsignaled otherwise. Its first and last arguments are the same as for *CreateMutex* — we always supply zeros. The second argument is the semaphore's initial value — a nonnegative integer. The third argument is something we don't see in POSIX: the semaphore's maximum value. This is presumably there for debugging and reliability purposes (if the program logic is such that a semaphore's value never should exceed some maximum value, this will give an error return if it does).

Dijkstra's P operation is supported using the *WaitFor* routines. When a thread returns from these routines after finding the semaphore in the signaled state, one is subtracted from the

semaphore's value. If it's now zero, the semaphore changes to the unsignaled state. The V operation is performed by *ReleaseSemaphore*, which is actually a bit more general than Dijkstra's version. Its second argument indicates how much to add to the semaphore's value. The third argument, if nonzero, is a pointer to an integer that, on return, holds the semaphore's previous value.

An important aspect of *WaitForMultipleObjects* is that it can be used with different types of objects. The following solution to the producer-consumer problem mixes semaphores with mutexes to allow multiple producers and consumers:

```

HANDLE pmutex = CreateMutex(0, FALSE, 0);
HANDLE cmutex = CreateMutex(0, FALSE, 0);
HANDLE empty = CreateSemaphore(0, B, B, 0);
HANDLE occupied = CreateSemaphore(0, 0, B, 0);
int nextin = 0;
int nextout = 0;

void produce(char item) {
    HANDLE harray[2] =
        {empty, pmutex};

    WaitForMultipleObjects(
        2, harray, TRUE,
        INFINITE);
    buf[nextin] = item;
    if (++nextin >= B)
        nextin = 0;
    ReleaseMutex(pmutex);
    ReleaseSemaphore(
        filled, 1, 0);
}

char consume( ) {
    char item;
    HANDLE harray[2] =
        {filled, cmutex};

    WaitForMultipleObjects(
        2, harray, TRUE,
        INFINITE);
    item = buf[nextout];
    if (++nextout >= BSIZE)
        nextout = 0;
    ReleaseMutex(cmutex);
    ReleaseSemaphore(
        empty, 1, 0);
    return(item);
}

```

So far we haven't seen anything that is much different from what POSIX supplies. We argued earlier that additional synchronization capabilities are needed over and above mutexes and semaphores. POSIX uses condition variables; Win-32 uses yet another kind of kernel object known as *event objects*. Like the other kernel objects we've discussed in this section, threads can wait for them to enter the signaled state. However, the state these objects are in is completely under programmer control; they have no use other than their state.

Event objects come in two forms: auto-reset and manual-reset. The distinction involves what happens when a thread finds them in the signaled state and returns from one of the *WaitFor* routines. Auto-reset objects automatically revert to the unsignaled state; manual-reset objects stay in the signaled state.

As an example, let's look at two more special cases of our event problem. In the first, suppose we have a switch controlled by on and off buttons. Threads arriving at the switch must wait if it's off, but are allowed to continue if it's on. This can be handled using manual-reset events:

```

HANDLE Switch = CreateEvent(0, TRUE, FALSE, 0);
// manual-reset event, initially unsignaled

```

```

void OnButtonPushed( ) {
    SetEvent(Switch);
}

void OffButtonPushed( ) {
    ResetEvent(Switch);
}

void WaitForSwitch( ) {
    WaitForSingleObject(Switch, INFINITE);
}

```

We create the event object by calling *CreateEvent*. Its first and last arguments are the same as in the other object-creation routines we've seen, and we supply zeros. Its second argument is TRUE if the event is to be manual-reset, FALSE if auto-reset. The third argument is the event's initial state, in this case *unsigaled*. Threads control the object's state by calling *SetEvent* to put it in the *sigaled* state and *ResetEvent* to put it in the *unsigaled* state. An additional routine, *PulseEvent*, is supplied. For manual-reset event objects, *PulseEvent* sets the object's state to *sigaled* just long enough to wake up all threads currently waiting for it, then resets it to *unsigaled*. For auto-reset event objects, *PulseEvent* sets the state to *sigaled* just long enough to release one thread. In either case, if no thread is waiting, nothing happens.

As a simple example of auto-reset events, let's modify our button-pushing example slightly so that one button, not two, toggles between on and off each time it's pressed.

```

HANDLE Switch = CreateEvent(0, FALSE, FALSE, 0);
    // auto-reset event, initially unsigaled

void ToggleButtonPushed( ) {
    SetEvent(Switch);    // release first thread to call, or
                        // first waiting thread to have called,
                        // WaitForPermission
}

void WaitForPermission( ) {
    WaitForSingleObject(Switch, INFINITE);
}

```

We finish our discussion of Win-32 synchronization by looking again at barriers. Without additional machinery, we'll have as difficult a time implementing them as we had with POSIX constructs. For example, a naive attempt at implementing a barrier might use a single manual-reset event object to represent threads waiting for the others to enter. When the last thread does enter, it uses *PulseEvent* to wake up the others. Of course, we have to protect access to the count of the number of threads that have entered with a mutex. So our code might look like

```

HANDLE mutex = CreateMutex(0, FALSE, 0);
HANDLE queue = CreateEvent(0, TRUE, FALSE, 0);
int count = 0;

```

```

void wait( ) {
    WaitForSingleObject(mutex, INFINITE);
    if (++count < n) {
        ReleaseMutex(mutex);
        WaitForSingleObject(queue, INFINITE);
    } else {
        count = 0;
        PulseEvent(queue);
        ReleaseMutex(mutex);
    }
}

```

But this program doesn't work. Suppose the next-to-the-last thread to enter the barrier (i.e., call *wait*) has just released the mutex. Before it gets a chance to call *WaitForSingleObject* and join the queue, the last thread enters the barrier. It quickly locks the mutex, increments *count*, finds that *count* is now *n*, and executes the *PulseEvent* statement. At this point, the previous thread finally gets around to calling *WaitForSingleObject*. Unfortunately, it's too late: the *PulseEvent* call has already taken place. There won't be another call, so this thread is stuck in the queue and no other thread will ever wake it up.

We can implement barriers, with the machinery we've discussed so far, by using a different, far more complicated and rather unintuitive approach. But one further operation on object handles (added by Microsoft a few years after the introduction of the other routines) lets us use our original, intuitive approach.

We need a way to put one object into its signaled state and, without the possibility that anything else happens in between, immediately commence to wait for another object to enter the signaled state. This new operation is *SignalObjectAndWait*; its operation is shown below in a correct barrier solution.

```

HANDLE mutex = CreateMutex(0, FALSE, 0);
HANDLE queue = CreateEvent(0, TRUE, FALSE, 0);
int count = 0;

void wait( ) {
    WaitForSingleObject(mutex, INFINITE);
    if (++count < n) {
        SignalObjectAndWait(mutex, queue, INFINITE, FALSE);
    } else {
        count = 0;
        PulseEvent(queue);
        ReleaseMutex(mutex);
    }
}

```

Here, in a single operation, we unlock the mutex and join the queue, thereby eliminating the problem with the first attempt. The first argument to *SignalObjectAndWait* is the handle of the object to be signaled. The second argument is the handle of the object for which we should wait until it is signaled. The third argument is the timeout. (The last argument indicates whether the call is "alertable," a topic we discuss in Section 5.2.3.1.)

2.2.4 THREAD SAFETY

Much of the standard Unix library code was developed before multithreaded programming was common. Though the Win-32 interface in Windows has the advantage of being developed with threads in mind, the interfaces for much of the standard libraries it must support, such as that used for the socket interface to networks, were designed before the age of threads. Our concern here is that a number of practices employed in these libraries coexist badly with multithreaded programs.

Our first example of such a practice is one we've already encountered: the system-call library, providing the interface between C (and C++) programs and the Unix operating system. The culprit is the means for reporting error codes through the global variable *errno*:

```
int IOfunc( ) {
    extern int errno;
    ...
    if (write(fd, buffer, size) == -1) {
        if (errno == EIO)
            fprintf(stderr, "IO problems ...\n");
        ...
        return(0);
    }
    ...
}
```

We have one global variable that is shared by all threads. Thus if two threads fail in system calls at about the same time, we'll have a conflict when each assumes *errno* holds its error code.

We might argue that the correct way to solve this problem is to change the interface, say by having system calls return their error codes, as in the POSIX threads routines, or have threads call a separate function to retrieve the error code, as in Win-32. However, so that we don't break all existing Unix programs, we must keep the interface the way it is. An imperfect approach that works well enough in this instance is to redefine *errno* as a function call:

```
#define errno    _errno( )
```

We then modify the system-call library to store a thread's error code in some nonglobal location. The *_errno* function would then return what's stored in that particular location. However, we still have the problem of finding a location for each thread's private instance of *errno*.

POSIX threads provides a general mechanism for doing this known as *thread-specific data*. Win-32 has essentially the same mechanism, known there as *thread-local storage*. The idea behind both is that we associate with each thread some storage for its private use. Then we provide some means so that when a thread refers to, say, *errno*, it gets the instance of *errno* stored in this private storage. Thus we have some sort of naming mechanism so that when different threads executing the same code refer to *errno*, each gets its private instance.

POSIX does not specify how this private storage is allocated. It might be pre-allocated when the thread is created or perhaps created on demand. However, to understand how the interface works, think of this storage as being implemented as an array of *void **, one array for each thread. To implement our naming mechanism we simply agree that *errno*, for example, appears at the same index within each array, as shown in Figure 2.7. Each instance of the data item is known as a *thread-specific data* item.

To settle on such an index (or *key*, as it is called in POSIX threads), one thread should call *pthread_key_create* to get a value for the key and store it someplace accessible to other threads, such as in a global variable. Each thread may then use the key to access its instance of the thread-specific data using the routines *pthread_setspecific* to set its value and *pthread_getspecific* to get its value:

```
int errno_key;    // global

void startup ( ) {
    // executed by only one thread
    ...
    pthread_key_create(&errno_key, 0);
    ...
}

int write(...) {    // wrapper in the system-call library
    int err = syscallTrap(WRITE, ...); // the actual trap into the
    kernel
    if (err)
        pthread_setspecific(errno_key, err);
    ...
}

#define errno pthread_getspecific(errno_key)
// make things easy to read and type

void IOfunc( ) {
    if (write(fd, buffer, size) == -1) {
        if (errno == EIO)
            fprintf(stderr, "IO problems ...\n");
        ...
        return(0);
    }
    ...
}
```

Note that *pthread_key_create* takes two parameters. The first is an output parameter indicating where the key should be stored. The second, if not zero, is the address of a cleanup routine (of type *void (*)(void *)*) called when each thread terminates that has used *pthread_setspecific* on the key.

This is useful in situations similar to the following in which storage is allocated when thread-specific data is initialized:

```
#define cred      pthread_getspecific(cred_key)
pthread_key_t cred_key;

mainline( ) {
    ...
    pthread_key_create(&cred_key, free_cred);
```

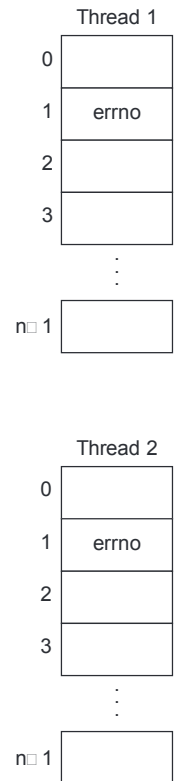


FIGURE 2.7
Thread-specific data.

```

    while (more_requests) {
        ...
        pthread_create(&thread, 0, server_start, request);
    }
}

void *server_start(void *req) {
    cred_t *credp;

    credp = (cred_t *)malloc(sizeof(cred_t));
    ...
    pthread_setspecific(cred_key, credp);
    ...
    handle_request(req);
    ...
    return(0);
}

handle_request(req_t req) {
    ...

    if (credentials_valid(req, cred))
        perform(req);

    ...
}

void free_cred(cred_t *credp) {
    ...
    free(credp);
}

```

Here we have a server application in which a new thread is created to handle each incoming request. Let's assume that along with each request some sort of credentials are established. Storage is allocated to hold these credentials, and a pointer to this storage is placed in the thread-specific data item whose key is given by *cred_key*. We've defined *cred* as a shortcut for accessing this data via *pthread_getspecific*. To make certain that the storage allocated to hold the credentials is released when the thread terminates, we define *free_cred* to be the cleanup function, via the call to *pthread_key_create*. Thus when each thread terminates, it calls *free_cred*.

Win-32's *thread-local-storage* mechanism is nearly identical to POSIX's thread-specific data.

Another thread-safety issue arises in the use of static local storage in the socket library used for networking. Below is a rough outline of *gethostbyname*, the routine used to get the address of a computer on the Internet:

```

struct hostent *gethostbyname(const char *name) {
    static struct hostent hostent;

```



```

... // lookup name; fill in hostent

return(&hostent);
}

```

Here we have the same sort of problem as with *errno*: there is a conflict if multiple threads call *gethostbyname* at once, since each will expect to see its result in the same location, the static local variable *hostent*. This problem is solved in the Win-32 version of the routine by the use of thread-local storage. POSIX threads treats this differently: a new version of *gethostbyname*, called *gethostbyname_r*, is defined in which the caller passes as an output argument the location at which the result should be stored.

A final problem is the use by multiple threads of library code that has shared access to data. A good example is the C standard I/O library, comprised of such routines as *printf*, *fgets*, and so forth. These routines share data structures describing and containing I/O buffers. Thus all threads calling *printf* buffer their output in the same buffer. To avoid problems, we must provide a mutex to synchronize access. Rather than making us do this manually, POSIX-threads and Win-32 implementations supply a special thread-safe version of the standard-I/O library providing appropriate synchronization along with the threads library.

2.2.5 DEVIATIONS

In our discussion so far, a thread's execution can be directly affected by others only when it's using a synchronization construct. The only thing others can do to a thread's execution is delay it; they can't force it to do something more drastic, such as terminate. In a number of situations, however, we'd like more to be possible; in particular, forcing another thread to terminate cleanly could be very useful.

Unix programmers have traditionally used Unix's *signal* mechanism to get a thread to deviate from its normal execution path, whether to handle some sort of event or to notify it of an exception. Win-32 doesn't provide a signal mechanism, but, as we argue below, it's not really necessary.

POSIX threads includes a *cancellation* mechanism by which one thread can request the clean termination of another. Win-32 does not include such a mechanism; it does provide techniques so that programmers can implement some of this functionality, though by no means as conveniently as in POSIX.

2.2.5.1 Unix Signals

Unix signals are a mechanism by which various sorts of actions trigger the operating system to interrupt a thread and immediately terminate or suspend it (and its process) or force it to put aside what it's doing, execute a prearranged handler for the action, then go back to what it was doing earlier. This is much the way that interrupts are handled in the operating system (see Chapter 3). Typical actions that can cause signals are typing special characters on the keyboard (such as Ctrl-C), timer expirations, explicit signals sent by other threads (via the *kill* system call) and program exceptions (arithmetic and addressing errors). For all but exception handling, signals are an artifact of the days of exclusively single-threaded programs. For exception handling, they are best thought of as a means for implementing language-specific mechanisms (something that C is sorely lacking).

The original intent of signals was to force the graceful termination of a process. For example, if you type the "interrupt" key, usually Ctrl-C, a SIGINT signal is sent to the currently running process. (To be precise, it's sent to the group of processes indicated as the foreground processes for the terminal.) If this process hasn't done anything special to deal with the signal, the operating system automatically and immediately terminates it.

A process might set up a handler to be invoked when such a signal is delivered, as shown below:

```
int main( ) {
    void handler(int);
    sigset(SIGINT, handler);

    /* long-running buggy code */
    ...
}

void handler(int sig) {
    /* perform some cleanup actions */
    ...
    exit(1);
}
```

The call to *sigset* in *main* causes *handler* to be registered as the signal handler for the SIGINT signal. Thus if you run this program and decide to terminate it early by typing Ctrl-C, a SIGINT signal is delivered to the process, causing it to put aside its current state and call *handler*; *handler* performs some cleanup actions (perhaps writing some key data to a file) and then terminates the process.

Signals can also be used as a means for communicating with a process, perhaps requesting it to do something:

```
computation_state_t state;

int main( ) {
    void handler(int);

    sigset(SIGINT, handler);

    long_running_procedure( );
}

long_running_procedure( ) {
    while (a_long_time) {
        update_state(&state);
        compute_more( );
    }
}

void handler(int sig) {
    display(&state);
}
```

In this code the *main* procedure establishes a handler for SIGINT, and then calls *long_running_procedure*, which might execute for several days. Occasionally the user of the

program wants to check to see what the program is doing, so she or he types Ctrl-C, which generates a signal causing the (single) thread to enter *handler*, print out that state (being maintained in *state* of some unspecified type *computation_state_t*), and then return back to *long_running_procedure*.

The above code might appear at first glance to be problem-free, but there is unsynchronized access to the data structure *state* within *update_state* and *display*. One might be tempted to use a mutex to synchronize access to *state*, but the situation here is different from what we have been assuming earlier: there is only one thread. Suppose that thread has locked the mutex within *update_state*, and then receives a signal and invokes *handler*. If it now attempts to lock the mutex within *display*, it will deadlock, since the mutex it is trying to lock is already locked (by itself).

Before we try to fix this problem, let's make sure we understand it. The issue is that the occurrence of a signal has forced the thread out of its normal flow of control and made it execute code (the signal handler) that interferes with what it was doing before the signal. This is similar to interthread synchronization issues, but we have only one thread. We discussed in Section 2.2.4 above the notion of thread-safe routines — procedures that a thread can execute without concern about interaction with other threads. Is there something analogous for signal handlers? There is, and such routines are said to be *async-signal safe*.

Operations on mutexes certainly are not *async-signal safe*, nor are dynamic-storage operations such as *malloc* and *free* — what if a thread in the middle of *malloc* gets interrupted by a signal whose handler calls *free*? The POSIX 1003.1 spec specifies well over sixty procedures that must be *async-signal safe* (including *fork*, *_exit*, *open*, *close*, *read*, *write*, and *sem_post*, but not *sem_wait*). Before reading on, you might think about what it takes for a procedure to be *async-signal safe*.

What can we do in our example to make *display* *async-signal safe*? We can make certain that any code that it might interfere with cannot be interrupted by a signal. In particular, we need to make certain that the thread cannot be interrupted by a signal while it's in *update_state*. This can be done as follows:

```
computation_state_t state;
sigset_t set;

int main( ) {
    void handler(int);

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigset(SIGINT, handler);

    long_running_procedure( );
}

long_running_procedure( ) {
    while (a_long_time) {
        sigset_t old_set;
        sigprocmask(SIG_BLOCK, &set, &old_set);
        update_state(&state);
        sigprocmask(SIG_SETMASK, &old_set, 0);
        compute_more( );
    }
}
```

```

    }
}

void handler(int sig) {
    display(&state);
}

```

Here, by using the routine *sigprocmask*, we block occurrences of SIGINT while the thread is inside *update_state*. This routine operates on signal sets, which are bit vectors representing sets of signals. We initialize such a set in *main*, first setting it to be the empty set, then adding SIGINT to it. Then, in *long_running_procedure*, we pass the set to *sigprocmask* as its second argument. The first argument, SIG_BLOCK, instructs the operating system to add the signals in the second argument to the current set of blocked signals, and to return the prior set of blocked signals in the set pointed to by the third argument. We call *sigprocmask* again to restore the set of blocked signals to its prior value.

Although this solves our problem, in many implementations of Unix *sigprocmask* is implemented as a system call and is thus somewhat expensive to use.⁹ Another issue is that the notion of blocking signals so as to prevent unwanted interaction with signal handlers is completely orthogonal to thread-based synchronization and adds a fair amount to the complexity of programs, making bugs more likely.

The reason this notion of asynchronous signals and signal handlers was necessary in Unix was that processes used always to be single-threaded. With multithreaded processes, there's no need to interrupt a thread to handle the event that caused the signal: we can simply dedicate a thread to handling the event.

How are signals handled in multithreaded processes? The first issue is that signals, as used in Unix, are sent to processes, not threads. In single-threaded processes it was obvious which thread would then handle the signal. But in multithreaded processes, it's not so clear. In POSIX threads the signal is delivered to a thread chosen at random.

Another issue is the set of blocked signals. Should one such set affect all threads in a process, or should each thread have its own set? Since threads add and remove signals to and from this set to protect their own execution, it makes sense for each thread to have its own blocked-signal set.

We can now clarify the POSIX rules for delivering signal to a multithreaded process: the thread that is to receive the signal is chosen randomly from the set of threads that do not have the signal blocked. If all threads have the signal blocked, then the signal remains pending until some thread unblocks it, at which point the signal is delivered to that thread.

With this definition of how signals are handled in multithreaded processes, we revisit our example.

```

computation_state_t state;
sigset_t set;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

int main( ) {
    pthread_t thread;

```

⁹ Solaris implements *sigprocmask* cheaply, not as a system call, and thus our example would work well on that system. Indeed, Solaris takes this a step further and blocks signals cheaply while threads are in a number of procedures such as *printf*, making them async-signal safe, even though POSIX doesn't require it.

```

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    pthread_sigmask(SIG_BLOCK, &set, 0);
    pthread_create(&thread, 0, monitor, 0);
    long_running_procedure( );
}

long_running_procedure( ) {
    while (a_long_time) {
        sigset_t old_set;
        pthread_mutex_lock(&mut);
        update_state(&state);
        pthread_mutex_unlock(&mut);
        compute_more( );
    }
}

void *monitor( ) {
    int sig;
    while (1) {
        sigwait(&set, &sig);
        pthread_mutex_lock(&mut);
        display(&state);
        pthread_mutex_unlock(&mut);
    }
    return(0);
}

```

In this variant of our example, the first thread blocks SIGINT. It uses *pthread_sigmask* rather than *sigprocmask*. The former is nominally for use by threads in multithreaded processes, though its effect is exactly the same as that of *sigprocmask*. This first thread then calls *pthread_create* to create a new thread, which starts execution in *monitor*. One property of *pthread_create* is that the new thread's set of blocked signals is copied from its creator. Thus the new thread starts in *monitor* with SIGINT blocked.

Inside *monitor* we have something new: a call to *sigwait*. This causes the calling thread to wait until a signal mentioned in the first argument occurs.¹⁰ Which signal did occur is then returned in storage pointed to by the second argument. This seems like (and is) a rather simple call, but it has a pretty powerful effect: signals such as SIGINT are no longer handled asynchronously, resulting in the interruption of some thread so that the handler can be called, but are now handled synchronously: a thread simply waits for the signal to be delivered, then deals with it. So now we can protect access to *state* simply by using a mutex.

Let's review what we've done with signals. The signal mechanism really has two parts: the generation of a signal, caused by the occurrence of some sort of event, and the delivery of the

¹⁰There is some ambiguity about what happens if a thread is waiting for a signal within *sigwait* and other threads have the signal unblocked. To insure that the only thread that deals with the signal is the one in *sigwait*, we block the signal for all threads.

signal to a thread. We've done nothing to modify the Unix treatment of the first. What we have done is to change the second drastically. No longer does the delivery of a signal force a thread to put aside what it's doing and call a signal handler. Unless we employ the appropriate signal blocking, this can be dangerous. Instead, we treat signal handling by threads as input processing: a thread simply waits for input (the notification that a signal has been delivered to it) and then deals with the signal. This is a simple programming model and lacks the additional complexity of traditional signal handling.

This makes it clear how Microsoft was able to avoid the notion of signals in its Win-32 interface. Since Win-32 applications are multithreaded, individual threads can be dedicated to dealing with the sort of events that would cause signals in Unix.

2.2.5.2 POSIX Cancellation

An annoying aspect of many programs is that, if you mistakenly start a lengthy operation, you can't cancel it in mid-operation — you must let it run to completion. The reason for this might simply be that the developer was inconsiderate of the program's users, but it could also be that the software packages used lacked sufficient support for early termination. This is particularly likely with multithreaded programs, which might contain a number of threads whose execution we wish to terminate, perhaps in response to a request to terminate the entire program, or because the chores being performed by these threads are no longer needed.

Of course, we can't simply terminate a thread at some arbitrary point in its execution, since doing so might leave a mutex locked or leave a data structure in some indeterminate state. Consider the following procedure, executed by a thread to insert an item at the beginning of a doubly linked list:

```
void insert(list_item_t *item) {
    pthread_mutex_lock(&list_mutex);
    item->backward_link = list_head;
    item->forward_link = list_head.forward_link;
    if (list_head.forward_link != 0)
        list_head.forward_link->backward_link = item;
    list_head.forward_link = item;
    pthread_mutex_unlock(&list_mutex);
}
```

If the thread executing this procedure were forced to terminate in the middle, not only would *list_mutex* be left in the locked state, but the item being inserted would end up partially inserted, with just some of the necessary links updated. Any future operations on the list are doomed to failure: not only will no other threads be able to lock the mutex, but the list itself is now malformed. If we adopt a mechanism allowing one thread to terminate another, we need to insure that procedures such as *insert* are atomic — once their execution begins, they must be allowed to run to completion.

Suppose the thread we wish to terminate has performed some actions whose effect must be undone if the thread does not run to completion. For example, the code below is executed by a thread that's gathering data and adding it to a list, using the above *insert* procedure.

```
list_item_t list_head;

void *GatherData(void *arg) {
```

```

    list_item_t *item;
    item = (list_item_t *)malloc(sizeof(list_item_t));
    GetDataItem(&item->value);
    insert(item);
    return(0);
}

```

The *GetDataItem* routine might take a long time to execute, so it might be necessary to force the termination of the thread while it's in that routine. But if the thread is terminated, we should let it clean up after itself: in particular, it should free the storage that's been allocated for the list item.¹¹ Thus another requirement we must make of any termination mechanism is that it should support such cleanup.

The POSIX feature providing all this is *cancellation*. Cancellation gives us what is essentially a means for one thread to raise an exception in another that will be handled when the target thread is at a safe point in its execution. (C, of course, has no exception mechanism. C++ does, but unfortunately there is no portable means for treating the POSIX-threads notion of cancellation as a C++ exception. However, see the description of the Brown C++ Threads package (Doeppner 1995).) Though POSIX requires a thread to terminate soon after incurring such an exception, it can first execute any number of cleanup routines.

One thread may request another to terminate by calling *pthread_cancel*, with the target thread's ID as the sole argument. The target is marked as having a *pending cancel* whose effect depends on the cancellation state set up in the target. Cancellation may be either *enabled* or *disabled*. If the latter, the cancel is at least temporarily ignored, though the thread is marked as having a pending cancel. If cancellation state is *enabled*, then what happens depends on whether the *cancel type* is set to asynchronous or deferred. If it's asynchronous, then the thread immediately acts on the cancel. Otherwise (*cancel type* is deferred), the thread ignores the cancel until it reaches a *cancellation point* in its execution, when it then acts on the cancel. The intent is that such cancellation points correspond to points in the thread's execution at which it is safe to act on the cancel.

When the thread does act on a cancel, it first walks through a stack of cleanup handlers it has previously set up, invoking each of them. It then terminates. The thread that executed *pthread_cancel* in the first place does not wait for the cancel to take effect, but continues on. If desired, it may call *pthread_join* to wait for the target to terminate.

Now for the details: threads have cancellation enabled and deferred when they are created. To change the cancellation state, a thread calls *pthread_setcancelstate* with two arguments. The first, an integer, is either *PTHREAD_CANCEL_DISABLE* or *PTHREAD_CANCEL_ENABLE*. The second is a pointer that, if not null, points to an integer into which the thread's cancellation state prior to the call is stored. To change the cancellation type, a thread calls *pthread_setcanceltype*, also with two arguments. The first, an integer, is either *PTHREAD_CANCEL_ASYNCHRONOUS* or *PTHREAD_CANCEL_DEFERRED*. The second is a pointer that, if not null, points to an integer into which the thread's cancellation type prior to the call is stored.

¹¹ The alert reader might argue that we wouldn't have this cleanup problem if the storage for *item* were allocated after *GetDataItem* returns. But let's assume that *malloc* is called first to insure that the resources required to hold the item exist before the data is obtained.

POSIX specifies that following routines are cancellation points:

- | | | |
|-----------------------|--------------------------|----------------|
| • aio_suspend | • open | • sigtimedwait |
| • close | • pause | • sigwait |
| • creat | • pthread_cond_wait | • sigwaitinfo |
| • fcntl ¹² | • pthread_cond_timedwait | • sleep |
| • fsync | • pthread_join | • system |
| • mq_receive | • pthread_testcancel | • tcdrain |
| • mq_send | • read | • wait |
| • msync | • sem_wait | • waitpid |
| • nanosleep | • sigsuspend | • write |

If a thread has a pending cancel and its cancellation type is deferred, then the cancel won't be acted upon unless the thread calls or is already in one of these cancellation points. Thus, for example, if a thread is waiting for input within *read*, on receipt of a cancel it quits the read call and immediately begins to act on the cancel.

Knowing what these cancellation points are, you should be careful with their use: you need to be certain that when a thread calls any such routine with cancellation enabled, it's safe for the thread to be terminated. Note that *pthread_mutex_lock* is not on the list. This allows you to use it freely without being certain that it's safe for the thread to terminate while it's waiting to lock a mutex. This is particularly important if a thread is holding one lock while waiting for another. Conversely, if there's a point in a program that you know is safe for cancellation, you can have the thread call *pthread_testcancel*, which does nothing if there is no pending cancel.

A thread's cancellation cleanup handlers are organized as a stack. A thread might push a cleanup handler on the stack when it enters a module for which cleanup is necessary, and then pop it off the stack on exit. When the thread acts on a cancel, it pops the handlers off the stack, invoking each one as it does, until the stack is empty, at which point the thread terminates.

To push a handler onto the stack, a thread calls *pthread_cleanup_push*, which takes two arguments. The first is a pointer to the routine being pushed as the cleanup handler, a routine whose type is *void (*)(void *)* (that is, a pointer to function with a single *void ** argument, returning *void*). The second argument is the value (of type *void **) that's passed to this routine when it's called. To pop the handler off the stack, the thread calls *pthread_cleanup_pop*, which takes one argument, called the *execute flag*. If it's zero, the handler is simply popped off the stack. If it's one, not only is it popped off the stack, but the thread also invokes it, passing it the argument registered in the original call to *pthread_cleanup_push*.

This is already a bit complicated, but there's more. The two routines *pthread_cleanup_push* and *pthread_cleanup_pop* are required to be "lexicographically scoped," meaning that each call to *pthread_cleanup_push* must have a unique matching call to *pthread_cleanup_pop*. In other words, you must treat them as if they were left and right parentheses and thus keep them balanced. The intent presumably was to make them clearly delineate their scope, but the effect can often be annoying, for instance when your module has multiple exit points at which you'd like to call *pthread_cleanup_pop* at each, but may not. Note that most implementations enforce this rule by implementing the routines as macros: *pthread_cleanup_push* contains an unmatched "{" and *pthread_cleanup_pop* contains an unmatched "}".

As an example of *pthread_cleanup_push* and *pthread_cleanup_pop*'s use, let's revisit our *GatherData* routine above. This time we add a cleanup handler.

¹² Only when F_SETLCKW is the command.


```

list_item_t list_head;

void *GatherData(void *arg) {
    list_item_t *item;
    item = (list_item_t *)malloc(sizeof(list_item_t));
    pthread_cleanup_push(free, item);
    GetDataItem(&item->value);
    pthread_cleanup_pop(0);
    insert(item);
    return(0);
}

```

We've added calls to *pthread_cleanup_push* and *pthread_cleanup_pop* so that, if the thread is cancelled while in *GetDataItem*, the storage it had allocated for *item* will be freed. Note that *malloc* and *insert* contain no cancellation points, so there's no danger, assuming the thread's cancellation type is deferred, of the thread acting on a cancel within them.

We know that *pthread_cond_wait* is a cancellation point. Normally a thread won't return from it until it has locked the mutex given as the second argument. What should it do if a cancel is acted upon within it? Consider the following:

```

pthread_mutex_lock(&mutex);
pthread_cleanup_push(CleanupHandler, argument);
while(should_wait)
    pthread_cond_wait(&cv, &mutex);

// . . . (code containing other cancellation points)

pthread_cleanup_pop(0);
pthread_mutex_unlock(&mutex);

```

What should *CleanupHandler* do? We want to make certain that if the thread acts on a cancel anywhere between the push and the pop, the mutex will end up unlocked. If we are assured the mutex is always locked on entry to *CleanupHandler*, we can simply unlock it there. But if, when a thread acts on a cancel within *pthread_cond_wait*, it does not lock the mutex, then there will be uncertainty within *CleanupHandler* whether the mutex is locked or unlocked. So, by insisting that a thread, when acting on a cancel within *pthread_cond_wait*, first lock the mutex, then perform its cleanup actions, we enable it to do the cleanup safely. Thus we can code our example as follows:

```

pthread_mutex_lock(&mutex);
pthread_cleanup_push(pthread_mutex_unlock, &mutex);
while(should_wait)
    pthread_cond_wait(&cv, &mutex);

// . . . (code containing other cancellation points)

pthread_cleanup_pop(1);          // unlock the mutex

```

Note that we take advantage of the execute flag to *pthread_cleanup_pop* by popping the handler off the stack and calling it at the same time.

We have one final word of caution concerning the use of cancellation in C++ programs. Consider the following C++ program fragment:

```
void tcode( ) {
    A a1;
    pthread_cleanup_push(handler, 0);
    subr( );
    pthread_cleanup_pop(0);
}

void subr( ) {
    A a2;
    pthread_testcancel( );
}
```

If the thread calling *tcode* has a pending cancel when it calls *subr*, it acts on it within *pthread_testcancel*. Since it has a cleanup handler in the calling scope (within *tcode*), it should leave *subr*, invoking the destructors for the objects local to *subr* (in particular, for *a2*). If it has no additional handlers other than the one pushed in *tcode*, it then terminates without calling the destructor for *a1*. So, the first thing to note is that if it is important for a destructor for a local object to be called as part of a thread's termination, that object must be in the scope of at least one of the cleanup handlers. You should also check to make certain that such automatic cleanup is indeed supported by the C++ system being used. If the above example is compiled and run with g++ (the Gnu C++ compiler), using a release up through at least 2.96, the destructors for neither *a1* nor *a2* are called — a result that is definitely a bug.

2.3 CONCLUSIONS

The purpose of this chapter was not just to teach you how to write a multithreaded program, but also to introduce various important operating-systems issues. The concerns include both how threads are implemented (which we cover in Chapter 5) and how multithreaded programming is used within operating systems (which we cover in Chapter 4). Experience gained from writing multithreaded programs, whether with POSIX threads, Win-32, or some other system, will help you appreciate much of what's discussed in following chapters on the inner workings of operating systems.

2.4 EXERCISES

Note: A number of programming exercises, with source code, are available at the book's web site, <http://www.wiley.com/college/Doeppner>.

1. Consider the following procedure, written in C:

```
unsigned int count = 0;
const int iterations = 1000000000; // one billion

unsigned int incr(void) {
    int i;
    for (i=0; i<iterations; i++)
```

```

        count++;
    return count;
}

```

Suppose *incr* is executed concurrently by ten threads, all sharing the global variable *count*. Each thread calls *incr*, then prints the value returned. If the execution of threads is not time-sliced, i.e., each thread runs until it voluntarily yields the processor, what values should be printed by each thread? Can you characterize the values printed if thread execution is time-sliced? Run such a program on your own computer. Is the execution of threads time-sliced? (Make sure that heavy optimization is turned off in the C compiler, otherwise it might figure out that the *for* loop can be replaced with a simple assignment!) (If your computer has more than one processor, this simple test won't work.)

2. Assuming that the execution of threads on your computer is time-sliced, what is the cost of this time-slicing in terms of the time required for the operating system to do it? To find out, write a procedure that does a certain amount of work performing a lengthy computation, such as the repeated addition in Exercise 1 above. Then compare the time taken by one thread doing all the work with the time taken by ten threads each doing one-tenth of the work. What can you say about the cost of time-slicing? How does the number of processors on your computer affect your result?
3. Redo exercise 1, above, but this time use a mutex to synchronize access to *count*. Employ one mutex shared by all threads; lock it before incrementing *count* and unlock it afterwards (i.e., lock and unlock the mutex inside the loop, not outside). Compare the running time with that of Exercise 1. What can you say about the cost of mutexes?
- *4. The following program, from (Peterson 1981), is a means, implemented without any special hardware or operating-system support, for two threads to have mutually exclusive access to a region of data or code. Each thread calls *peterson* with its own ID, which is either 0 or 1. It assumes that each thread is running on a separate processor and that the memory holding the shared variables is *strictly coherent*, meaning that a store into a memory location by one processor will be seen by the other processor if it does a load from that location immediately thereafter.

```

unsigned int count = 0;                // shared

void *peterson(void *arg) {
    int me = (int)arg;
    static int loser;                  // shared
    static int active[2] = {0, 0};    // shared
    int other = 1-me;                 // private
    int i;

    for (i=0; i<iterations; i++) {
        active[me] = 1;
        loser = me;
        while (loser == me && active[other])
            ;
    }
}

```

```

        count++;    // critical section
        active[me] = 0;
    }
    return(0);
}

```

The variables *loser* and *active* are shared by both threads, the other variables are private to each of the threads (i.e., each has its own copy). The idea is that the *active* array is a pair of flags, one for each thread, that the thread sets to indicate that it has entered the “critical section,” in which the shared variable *count* is incremented. Each thread, before entering the critical section, sets its entry in *active*, then checks the other thread’s entry to see if that thread is already in the critical section. If so, the thread waits, by repeatedly checking the array. Of course, it is possible that both threads attempt to enter at the same time. This is where the *loser* variable comes in: after setting its *active* entry to 1, a thread sets *loser* to its own ID. If both threads are attempting to enter the critical section, *loser*’s value will be the ID of just one of the two threads. That thread must yield to the other.

- a. Write a program that tests whether this algorithm works. To do so, have two threads call *peterson*, with *iterations* set to a large number, such as two billion. If the code correctly provides mutual exclusion, then the final value of *count* will be twice the value of *iterations*. If not, it will be less. Explain why.
 - b. Modern shared-memory multiprocessor computers (including computers employing multicore chips) do not have strictly coherent memory. Instead, the effect of stores to memory is delayed a bit: when one processor stores a value into a memory location, loads by other processors might not retrieve the new value until after a few clock cycles (though loads by the processor performing the store will retrieve the new value, since it is in that processor’s memory cache). Explain why *peterson* does not always work correctly on modern share-memory multiprocessors.
- *5. The following solution to the producer-consumer problem is from (Lamport 1977). It works for a single producer and a single consumer, each running on a separate processor on a shared-memory multiprocessor computer.

```

char buf[BSIZE];
int in = 0;
int out = 0;

void producer(char item) {
    while(in-out == BSIZE)
        ;
    buf[in%BSIZE] = item;
    in++;
    return;
}

char consumer(void) {

```

```

    char item;
    while(in-out == 0)
        ;
    item = buf[out%BSIZE];
    out++;
    return(item);
}

```

It is easy to see how this works if you first assume that integers can be of arbitrary size. Thus *in* counts the number of bytes produced and *out* counts the number of bytes consumed. The difference between the two, $in - out$, is the number of bytes in the buffer. The solution continues to work even if integers are finite, as long as they can hold values larger than BSIZE (you might try to verify this, but you may assume it is correct for the questions below).

- a. Does this solution to the producer-consumer problem work, even if memory has the delayed-store property of Exercise 4b?
 - b. Some shared-memory multiprocessors have even weaker memory semantics. Normally we expect that if a processor stores into location A, then into location B, and then another processor loads from A and then B, then if the second processor observes the store into B, it must also have observed the store into A. In other words, we expect the order of stores by one processor to be seen by the other processors. However, consider a system with two memory units. A is in one of them; B is in the other. It may be that the unit holding A is extremely busy, but the one holding B is not. Thus it is conceivable that the store into A will be delayed longer than the store into B, and thus other processors will see B updated before A. Suppose our computer has this “reordered store” property. Will Lamport’s solution to the producer-consumer problem necessarily work on such a computer? Explain.
- * 6. Consider the following implementation of *pthread_cond_wait* and *pthread_cond_signal*. Assume that each condition variable (an item of type *pthread_cond_t*) contains a semaphore that is initialized to 0.

```

pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m) {
    pthread_mutex_unlock(m);
    sem_wait(c->sem);
    pthread_mutex_lock(m);
}

pthread_cond_signal(pthread_cond_t *c) {
    sem_post(c->sem);
}

```

Is this implementation correct? (We are concerned only with the “normal” functioning of condition variables; we are not concerned about cancellation and interaction with signal handlers.) Explain. Does this give you any insight into why operations on condition variables are specified as they are?

2.5 REFERENCES

- Dijkstra, E. W. (1968). The Structure of the “THE”-Multiprogramming System. *Communications of the ACM* **11**(5): 341–346.
- Dijkstra, E. W. (undated: early 1960s). Multiprogramming en de X8, at <http://userweb.cs.utexas.edu/users/EWD/ewd00xx/EWD57.PDF>
- Doeppner, T. W. (1987). *Threads: A System for the Support of Concurrent Programming*, Brown University, at <http://www.cs.brown.edu/~twd/ThreadsPaper.pdf>
- Doeppner, T. W. (1996). *The Brown C++ Threads Package*, Brown University, at <http://www.cs.brown.edu/~twd/c++threads.pdf>
- Lamport, L. (1977). Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* **SE-3**(2): 125–143.
- Peterson, G. L. (1981). Myths About the Mutual Exclusion Problem. *Information Processing Letters* **12**(3): 115–116.

3.1	Context Switching	3.3.1	Best-Fit and First-Fit Algorithms
3.1.1	Procedures	3.3.2	Buddy System
3.1.1.1	Intel x86 Stack Frames	3.3.3	Slab Allocation
3.1.1.2	SPARC Stack Frames	3.4	Linking and Loading
3.1.2	Threads and Coroutines	3.4.1	Static Linking and Loading
3.1.3	System Calls	3.4.2	Shared Libraries
3.1.4	Interrupts	3.5	Bootting
3.2	Input/Output Architectures	3.6	Conclusions
3.3	Dynamic Storage Allocation	3.7	Exercises
		3.8	References

This chapter covers a number of basic concepts that, though not unique to operating systems, are essential for much of what comes later. We start with the architectural issues involved in context switching: subroutine, thread, user/privileged, and normal/interrupt-mode switching. We then discuss some other architectural issues: memory caches and basic I/O architecture. Finally, we briefly go over the software concepts of dynamic storage allocation and of program linking and loading.

We use the somewhat nebulous term “context” to mean the setting in which execution is currently taking place. This setting determines what information, in the form of storage contents, is available. For example, when a thread is executing within a procedure, it has available to it the procedure’s local variables as well as all its process’s global variables. When a thread has invoked a method of some object, then not only are local and global variables available, but so are instance variables. When a thread makes a system call, the processor switches from user mode to privileged mode and the thread switches from user context to system context — information within the operating system is now available to it. Note that these contexts overlap: for instance, a thread in a user context or the system context enters and exits numerous procedure contexts.

We also think of a thread itself as a context. It’s the processor that’s in a thread context and can switch among different contexts. If an interrupt occurs, the processor switches from the thread context (or perhaps another interrupt context) into an interrupt context. Thus when a process starts execution in *main*, the processor is running in a thread context, which is running in a user context within the context of *main*.

In the remainder of this section we examine the mechanisms for switching among contexts, focusing on switches from procedure to procedure, from thread to thread, from user to system and back, and from thread to interrupt and back.

3.1 CONTEXT SWITCHING

3.1.1 PROCEDURES

The following code illustrates a simple procedure call in C:

```
int main( ) {
    int i;
    int a;

    . . .

    i = sub(a, 1);

    . . .

    return(0);
}

int sub(int x, int y) {
    int i;
    int result = 1;
    for (i=0; i<y; i++)
        result *= x;
    return(result);
}
```

The purpose of the procedure *sub* is pretty straightforward: it computes x^y . How the context is represented and is switched from that of *main* to that of *sub* depends on the architecture. The context of *main* includes any global variables (none in this case) as well as its local variables, *i* and *a*. The context of *sub* also includes any global variables, its local variables, *i* and *result*, and its arguments, *x* and *y*. On most architectures, global variables are always found at a fixed location in the address space, while local variables and arguments are within the current stack frame.

3.1.1.1 Intel x86 Stack Frames

Figure 3.1 is a general depiction of the stack on the Intel x86 architecture. Associated with each incarnation of a subroutine is a *stack frame* that contains the arguments to the subroutine, the *instruction pointer* (in register *eip*) of the caller (i.e., the address to which control should return when the subroutine completes), a copy of the caller's *frame pointer* (in register *ebp*), which links the stack frame to the previous frame, space to save any registers modified by the subroutine, and space for *local variables* used by the subroutine. Note that these frames are of variable size — the size of the space reserved for local data depends on the subroutine, as does the size of the space reserved for registers.

The frame pointer register *ebp* points into the stack frame at a fixed position, just after the saved copy of the caller's instruction pointer (note that lower-addressed memory is towards the bottom in the figure). The subroutine does not change the value of the frame pointer, except setting it on entry to the subroutine and restoring it on exit. The stack pointer *esp* always points to the last item on the stack — new allocations, say for arguments to be passed to the next procedure, are performed here. Note that register *eax*, used for the return value of procedures, is expected to be modified across calls and is thus not saved.

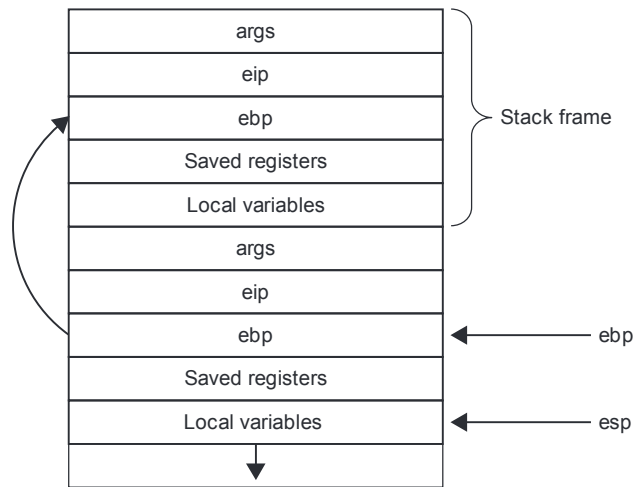


FIGURE 3.1 Intel x86 stack frames.

The picture above is idealized: not all portions of the stack frame are always used. For example, registers are not saved if the subroutine doesn't modify them, the frame pointer is not saved if it's not used, etc. For more details, see the Intel architecture manuals (<http://www.intel.com/design/processor/manuals/253665.pdf>).

Here's possible unoptimized assembler code for the above C code:¹

```
main:
    ; enter main, creating a new stack frame
    pushl %ebp                ; Push frame pointer onto the stack;
                                ; this means that the contents of the
                                ; stack-pointer register (esp) are
                                ; reduced by 4 so that esp points to
                                ; next lower word in the stack, then
                                ; the contents of the frame-pointer
                                ; register (ebp) are stored there.
    movl %esp, %ebp           ; Set frame pointer to point to new
                                ; frame: the address of the current end
                                ; of the stack (in register esp) is
                                ; copied to the frame-pointer register.
    pushl %esi                ; Save esi register: its contents are
                                ; pushed onto the stack.
    pushl %edi                ; Save edi register: its contents are
                                ; pushed onto the stack.
    subl $8, %esp             ; Create space for local variables (i
```

¹ We use the syntax of the Gnu assembler. It's similar to that of the Microsoft assembler (masm) except in the order of arguments: in the Gnu assembler, the first argument is the source and the second is the destination; in the Microsoft assembler, the order is reversed.

```

; and a): this is done by reducing the
; stack-pointer register (esp) by 8,
; thus lowering the end of the stack
; by 2 words.

... ; other code that uses esi and edi registers

; set up the call to sub

; the arguments are pushed onto the stack in reverse order
pushl $1          ; Push argument two onto the stack.
movl  -4(%ebp), %eax ; Put local variable a (argument
                    ; number one) into eax register.
pushl %eax        ; Push eax onto stack, thus pushing
                    ; argument two onto the stack.
call  sub         ; Call the subroutine sub: push eip
                    ; (instruction pointer) onto stack and
                    ; then set eip to address of sub:
                    ; thus the program branches to sub.
addl  $8, %esp    ; The call to sub has returned;
                    ; pop arguments from stack by simply
                    ; increasing the stack-pointer
                    ; register's value by 8: thus the two
                    ; words that were at the "top" of the
                    ; stack are no longer on the stack.
movl  %eax, -8(%ebp) ; Store sub's return value into i:
                    ; this value was returned in register
                    ; eax; store this value in the second
                    ; word from the beginning of the stack
                    ; frame (that is, 8 bytes less than
                    ; the address contained in register
                    ; ebp).

...

movl  $0, %eax    ; Set main's return value to 0.
; leave main, removing stack frame
popl  %edi        ; Restore edi register: at this point
                    ; the word at the end (top) of the
                    ; stack is the saved contents of
                    ; register edi; the popl instruction
                    ; copies this value into register edi,
                    ; then increases the value of esp, the
                    ; stack-pointer register, by 4, thereby

```

```

                                ; removing the word from the stack.
popl    %esi                    ; Restore esi register.
movl    %ebp, %esp              ; Restore stack pointer: recall that
                                ; its original value was saved in the
                                ; frame-pointer register (ebp).
popl    %ebp                    ; Restore frame pointer.
ret                                     ; Return: pop end of stack into eip
                                ; register, causing jump back to
                                ; caller.

sub:
    ; enter sub, creating new stack frame
    pushl %ebp                  ; Push frame pointer.
    movl  %esp, %ebp            ; Set frame pointer to point to new
                                ; frame.
    subl  $8, %esp              ; Allocate stack space for local
                                ; variables.

                                ; At this point the frame pointer
                                ; points into the stack at the
                                ; saved copy of itself; the two
                                ; arguments to sub are at locations
                                ; 8 and 12 bytes above this location.

    ; body of sub
    movl  $1, -4(%ebp)          ; Initialize result: the value one is
                                ; stored in the stack frame at the
                                ; location reserved for result.
    movl  $0, -8(%ebp)          ; Initialize i: the value zero is
                                ; stored in the stack frame at the
                                ; location reserved for i.
    movl  -4(%ebp), %ecx         ; Put result in ecx register.
    movl  -8(%ebp), %eax         ; Put i in eax register.
beginloop:
    cmpl  12(%ebp), %eax         ; Compare i with y (i is in the eax
                                ; register; y is in the stack 12
                                ; bytes above the location pointed
                                ; to by the frame-pointer register
                                ; (ebp)).
    jge   endloop               ; Exit loop if i >= y ("jge" means
                                ; "jump on greater than or equal"
                                ; and refers to the result of the
                                ; previous comparison).
    imull 8(%ebp), %ecx          ; Multiply result (in ecx register) by

```

```
                                ; x ("imull" means "integer multiply").
addl    $1, %eax                ; Add 1 to i.
jmp     beginloop              ; Go to beginning of for loop.
endloop:
movl    %ecx, -4(%ebp)          ; Store ecx back into result.
movl    -4(%ebp), %eax          ; Load result in return register (eax).
; leave sub, removing stack frame
movl    %ebp, %esp              ; Pop local variables off of stack.
popl    %ebp                    ; Pop frame pointer.
ret                                     ; Return: pop word at end of stack
                                ; into eip register, causing jump
                                ; back to caller.
```

3.1.1.2 SPARC Stack Frames

Note: this section may be skipped by readers with no interest in the SPARC architecture. It is provided as an example to show the differences between a RISC (reduced-instruction-set computer) architecture and a CISC (complex-instruction-set computer) architecture.

Sun’s SPARC (Scalable Processor ARChitecture) is a RISC (reduced-instruction-set computer). Its architecture manual can be found at <http://www.sparc.com/standards/V8.pdf>. We cover here just the details of the SPARC architecture that are relevant to subroutine-calling conventions. As shown in Figure 3.2, the SPARC has nominally 32 registers arranged in four groups of eight — *input registers*, *local registers*, *output registers*, and *global registers*. Two of the input registers serve the special purposes of a *return address register* and a *frame pointer*, much like the corresponding registers on the Intel x86. One of the output registers is the *stack pointer*. Register 0 (of the global registers) is very special — when read it always reads 0 and when written it acts as a sink.

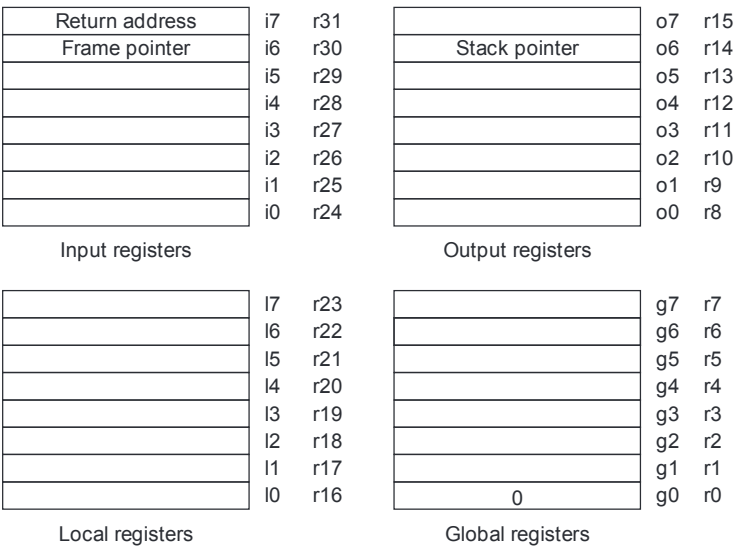


FIGURE 3.2 SPARC registers.

As its subroutine-calling technique the SPARC uses *sliding windows*: when a subroutine is called, the caller's output registers become the callee's input registers. Thus the register sets of successive subroutines overlap, as shown in Figure 3.3.

Any particular implementation of the SPARC has a fixed number of register sets (of eight registers apiece) — seven in Figure 3.3. As long as we do not exceed the number of register sets, subroutine entry and exit are very efficient — the input and local registers are effectively saved (and made unavailable to the callee) on subroutine entry, and arguments (up to six) can be efficiently passed to the callee. The caller just puts outgoing arguments in the output registers and the callee finds them in its input registers.

Returning from a subroutine involves first putting the return value in a designated input register, i0. In a single action, control transfers to the location contained in i7, the return address register, and the register windows are shifted so that the caller's registers are in place again.

However, if the nesting of subroutine calls exceeds the available number of register sets, then subroutine entry and exit are not so efficient. The register windows must be copied to an x86-like stack (see Figure 3.4). As implemented on the SPARC, when an attempt is made to nest subroutines more deeply than the register windows can handle, a *window-overflow trap* occurs and the operating system must copy the registers to the thread's stack and reset the windows (privileged instructions are required to access the complete set of on-chip registers). Similarly, when a subroutine

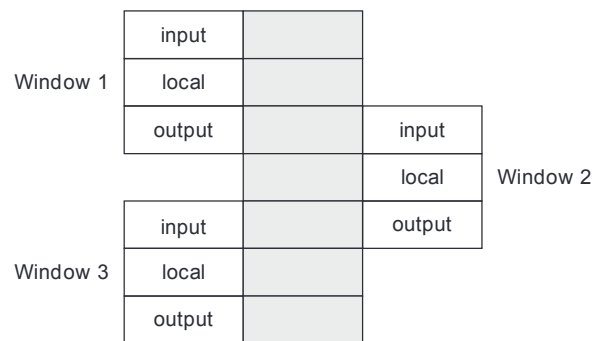


FIGURE 3.3 SPARC sliding windows.

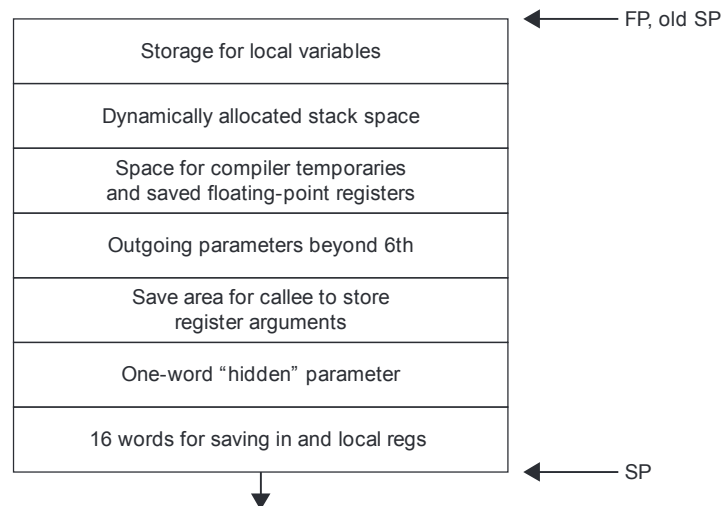


FIGURE 3.4 SPARC stack frame.

return encounters the end of the register windows, a *window-underflow trap* occurs and the operating system loads a new set of registers from the values stored on the thread's stack.

The actual code generated for calling, entering, and exiting a procedure is:

```
main:
    ld      [%fp-8], %o0      ! Put local variable a into o0
                                ! register as 1st param.
    mov     1, %o1            ! Put 2nd parameter into o1 register.
    call    sub               ! Call sub.
    nop
    st      %o0, [%fp-4]      ! Store result into local variable i.
    ...

sub:
    save    %sp, -64, %sp     ! Push a new stack frame.
    ...

    ret                                ! Return to caller.
    restore                                ! Pop frame off stack (in delay
                                ! slot).
```

The first step in preparing for a subroutine call is to put the outgoing parameters into the output registers. The first parameter *a*, from our original C program, is a local variable and is found in the stack frame. The second parameter is a constant. The *call* instruction merely saves the program counter in *o7* and then transfers control to the indicated address.

In the subroutine, the *save* instruction creates a new stack frame and advances the register windows. It creates the new stack frame by taking the old value of the stack pointer (in the caller's *o6*), subtracting from it the amount of space that is needed (64 bytes in this example), and storing the result into the callee's stack pointer (*o6* of the callee). At the same time, it also advances the register windows, so that the caller's output registers become the callee's input registers. If there is a window overflow, the operating system takes over.

Inside the subroutine, the return value is computed and stored into the callee's *i0*. The *restore* instruction pops the stack and backs down the register windows. Thus what the callee put in *i0* is found by the caller in *o0*. The *ret* instruction causes a jump to the program counter, saved in *i7*.

Note that the order of the *ret* and *restore* instructions is reversed from what one might expect: instructions that cause a transfer of control have a delayed effect, since by the time the transfer takes place, the processor has already fetched and executed the next instruction. Thus the *restore* instruction is placed after the *ret* instruction (in its *delay slot*) and is executed before control returns to the caller. See the SPARC architecture manual for details (SPARC 1992).

3.1.2 THREADS AND COROUTINES

Our notion of threads is that they are independent of one another and, except for synchronization and cancellation, don't directly control one another's execution. However, within the implementation of a threads package it may be necessary for one thread to transfer control directly to another, yielding the processor to the other without invoking the operating system. This sort of direct transfer of control from one thread to another is known as *coroutine* linkage. A thread's

context is mainly represented by its stack, but its register state is also important, such as the current value of the stack pointer. Let's assume that this and other components of its context are stored in a data structure we call the *thread control block*.

Switching between thread contexts turns out to be very straightforward, though it can't be expressed in most programming languages. We have an ordinary-looking subroutine, *switch*. A thread calls it, passing the address of the control block of the thread to whose context we wish to switch. On entry to the subroutine the caller's registers are saved. The caller saves its own stack pointer (SP) in its control block. It then fetches the target thread's stack pointers from its control block and loads it into the actual stack pointer. At this point, we have effectively switched threads, since we are now executing on the target thread's stack. All that has to be done now is to return — the return takes place on the target thread's stack.

To make this easier to follow, let's work through what happens when some thread switches to our original thread: it will switch to the original thread's stack and execute a return, in the context (on the stack) of the original thread. Thus, from the point of view of the original thread, it made a call to *switch*, which didn't appear to do much but took a long time.

Assuming that the global variable *CurrentThread* points to the control block of the current thread, the code for *switch* looks like:

```
void switch(thread_t *next_thread) {
    CurrentThread->SP = SP;
    CurrentThread = next_thread;
    SP = CurrentThread->SP;
    return;
}
```

On an Intel x86, the assembler code would be something like the following. Assume SP is the offset of the stack pointer within a thread control block.

```
switch:
    ; enter switch, creating new stack frame
    pushl %ebp                ; Push frame pointer.
    movl %esp, %ebp           ; Set frame pointer to point to new
                                ; frame.
    pushl %esi                ; Save esi register
    movl CurrentThread, %esi   ; Load address of caller's
                                ; control block.
    movl %esp, SP(%esi)        ; Save stack pointer in control block.
    movl 8(%ebp), CurrentThread ; Store target thread's
                                ; control block address
                                ; into CurrentThread.
    movl CurrentThread, %esi    ; Put new control-block
                                ; address into esi.
    movl SP(%esi), %esp        ; Restore target thread's stack pointer.
                                ; we're now in the context of the target thread!
    popl %esi                  ; Restore target thread's esi register.
    popl %ebp                  ; Pop target thread's frame pointer.
    ret                        ; Return to caller within target thread.
```

Performing such a coroutine switch on the SPARC architecture requires a bit more work, since at the moment the *switch* routine is called, the on-chip register windows contain the calling thread's registers. The operating system must be called explicitly (via a system call) to produce the effect of a window-overflow trap, forcing the register windows to be written to the calling thread's stack. Then when *switch* returns in the context of the target thread, a window-underflow trap occurs and the target thread's registers are loaded from its stack.

Thus the assembler code on the SPARC would be something like the following, assuming *CurrentThread* is kept in global register *g0*:

```
switch:
    save    %sp, -64, %sp      ! Push a new stack frame.
    t       3                  ! Trap into the OS to force window
                                ! overflow.
    st       %sp, [%g0+SP]      ! Save CurrentThread's SP in control
                                ! block.
    mov      %i0, %g0           ! Set CurrentThread to be target
                                ! thread.
    ld       [%g0+SP], %sp      ! Set SP to that of target thread
    ret                                ! return to caller (in target thread's
                                ! context).
    restore                                ! Pop frame off stack (in delay slot).
```

3.1.3 SYSTEM CALLS

System calls involve the transfer of control from user code to system (or kernel) code and back again. Keep in mind that this does not involve a switch between different threads — the original thread executing in user mode merely changes its execution mode to kernel (privileged) mode. However, it is now executing operating-system code and is effectively part of the operating system.

Most systems provide threads with two stacks, one for use in user mode and one for use in kernel mode. Thus when a thread performs a system call and switches from user mode to kernel mode, it also switches from its user-mode stack to its kernel-mode stack.

For an example, consider a C program running on a Unix system that calls *write* (see Figure 3.5). From the programmer's perspective, *write* is a system call, but a bit more work needs to be done before we enter the kernel. *Write* is actually a routine supplied in a special library of (user-level) programs, the C library. *Write* is probably written in assembler language; the heart of it is some instruction that causes a trap to occur, thereby making control enter the operating system. Prior to this point, the thread had been using the thread's user stack. After the trap, as part of entering kernel mode, the thread switches to using the thread's kernel stack. Within the kernel our thread enters a fault-handler routine that determines the nature of the fault and then calls the handler for the *write* system call.

3.1.4 INTERRUPTS

When an interrupt occurs, the processor puts aside the current context (that of a thread or another interrupt) and switches to an interrupt context. When the interrupt handler is finished, the processor generally resumes the original context. Interrupt contexts require stacks; which stack is used? There are a number of possibilities: we could allocate a new stack each time an interrupt occurs, we could have one stack that is shared by all interrupt handlers, or the interrupt handler could borrow a stack from the thread it is interrupting.

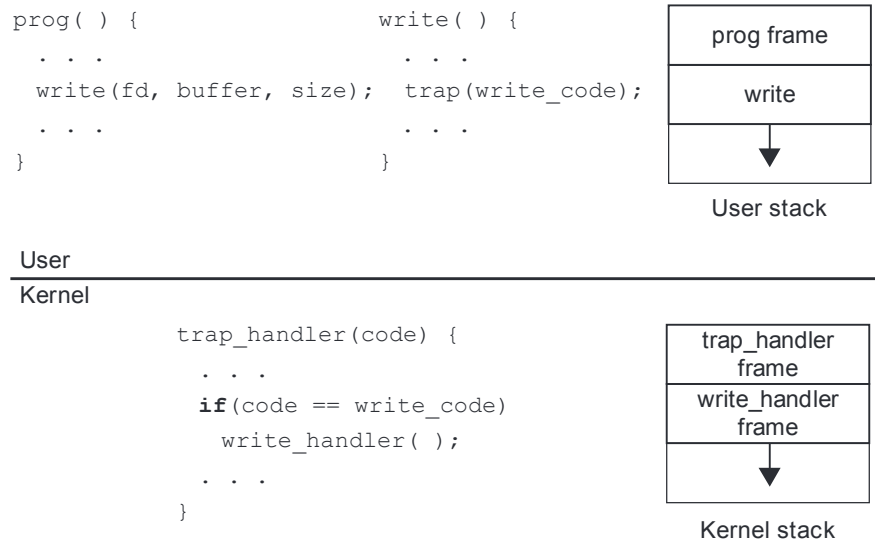


FIGURE 3.5 System-call flow.

The first technique, allocating a stack, is ruled out for a number of reasons, not least that it is too time-consuming, though a variant is used in Solaris, as discussed below. Both the latter two approaches are used. A single system-wide interrupt stack was used on DEC's VAX computers;² in most other architectures the interrupt handler borrows a stack (the kernel stack) from the thread that was interrupted (see Figure 3.6). If another interrupt handler was interrupted, its context is saved on the current stack, and that stack continues to be used for the new interrupt handler.

On those systems on which multiple interrupt handlers use the same stack, perhaps one borrowed from a thread, interrupt handlers must execute differently from normal thread execution. Because there is just one shared stack, there is just one interrupt context. We cannot suspend the execution of one handler and resume the execution of another: the handler of the most recent interrupt must run to completion. Then, when it has no further need of the stack, the handler of the next-most-recent interrupt completes its execution, and so forth.

That the execution of an interrupt handler cannot be suspended is significant because it means that interrupt handling cannot yield to anything other than higher-priority interrupts. For example, on a uniprocessor system, if interrupt handler 2 interrupts interrupt handler 1 and then wakes up a high-priority thread that must immediately respond to an event, the thread cannot run until all interrupt handling completes, including that of interrupt handler 1.

If we could somehow give each interrupt handler its own stack, we wouldn't have this problem. Solaris, as well as some real-time systems, has preallocated stacks for each possible interrupt level. Thus each has its own independent context and can yield to other processing, just as threads can. We discuss this in greater detail in Chapter 5. Most other general-purpose systems do not support this style of architecture.

Another approach to getting interrupt handlers to yield to other execution is that the handler places a description of the work that must be done on a queue of some sort, then arranges for it to be done in some other context at a later time. This approach, which is used in many systems including Windows and Linux, is also discussed in Chapter 5.

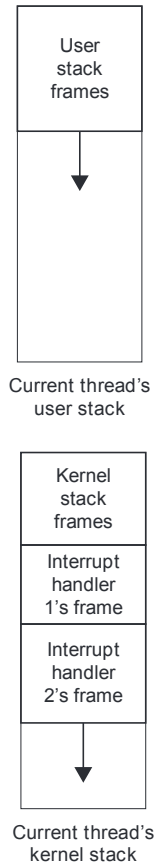


FIGURE 3.6 Interrupt handlers and stacks.

² Multiprocessor versions of the VAX had one interrupt stack per processor.

An important property of interrupts is that they can be *masked*, i.e., temporarily blocked. If an interrupt occurs while it is masked, the interrupt indication remains pending; once it is unmasked, the processor is interrupted. How interrupts are masked is architecture-dependent, but two approaches are common. One approach is that a hardware register implements a bit vector — each bit represents a class of interrupts. If a particular bit is set, then the corresponding class of interrupts is masked. Thus the kernel masks interrupts by setting bits in the register. When an interrupt does occur, the corresponding mask bit is set in the register and cleared when the handler returns — further occurrences of that class of interrupts are masked while the handler is running.

What's more common are hierarchical interrupt levels. Each particular device issues interrupts at a particular level. The processor masks interrupts by setting an *interrupt priority level* (IPL) in a hardware register: all devices whose interrupt level is less than or equal to this value have their interrupts masked. Thus the kernel masks a class of interrupts by setting the IPL to a particular value. When an interrupt does occur and the handler is invoked, the current IPL is set to that of the device, and restored to its previous value when the handler returns.

3.2 INPUT/OUTPUT ARCHITECTURES

In this section we give a high-level, rather simplistic overview of common I/O architectures. Our intent is to provide just enough detail to discuss the responsibilities of the operating system in regard to I/O, but without covering the myriad arcane details of device management. To do this, we introduce a simple I/O architecture we have used in the past at Brown University for operating system projects.

A very simple architecture is the *memory-mapped* architecture: each device is controlled by a controller and each controller contains a set of registers for monitoring and controlling its operation (see Figure 3.7). These registers appear to the processor to occupy physical memory locations. In reality, however, each controller is connected to a *bus*. When the processor wants to access or modify a particular location, it broadcasts the address on the bus. Each controller listens for a fixed set of addresses and, when one of its addresses has been broadcast, attends to what the processor wants to have done, e.g., read the data at a particular location or modify the data at a particular location. The memory controller, a special case, passes the bus requests to the actual primary memory. The other controllers respond to far fewer addresses, and the effect of reading and writing is to access and modify the various controller registers.

There are two categories of devices, *programmed I/O* (PIO) devices and *direct memory access* (DMA) devices. PIO devices do I/O by reading or writing data in the controller registers one byte or word at a time. In DMA devices the controller itself performs the I/O: the processor puts a description of the desired I/O operation into the controller's registers, then the controller takes over and transfers data between a device and primary memory.

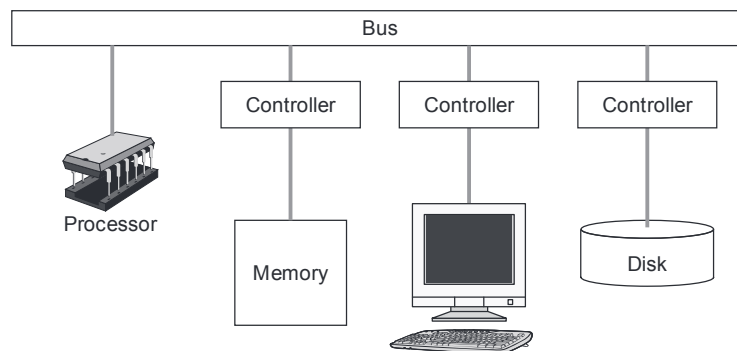


FIGURE 3.7 Simple I/O architecture.

Our architecture supports both PIO and DMA devices. The default configuration has one PIO device (a terminal) and one DMA device (a disk). Each PIO device has four registers: *Control*, *Status*, *Read*, and *Write*, each one byte in length (see Figure 3.8). Each DMA device also has four registers: *Control*, *Status*, *Memory Address*, and *Device Address* (Figure 3.9). The control and status registers are each one byte long; the others are four bytes long (they hold addresses). Certain bits of the control registers are used to start certain functions, as shown in Figure 3.8 and Figure 3.9. Bits of the status registers are used to indicate whether the associated controller is ready or busy.

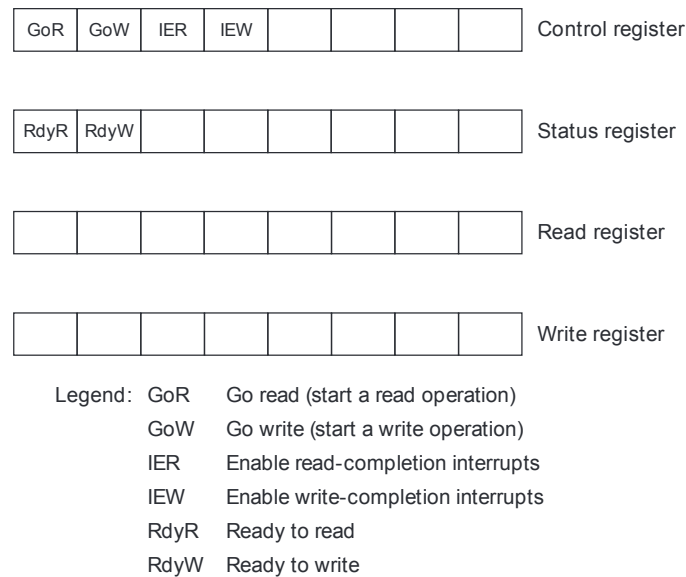


FIGURE 3.8 PIO registers.

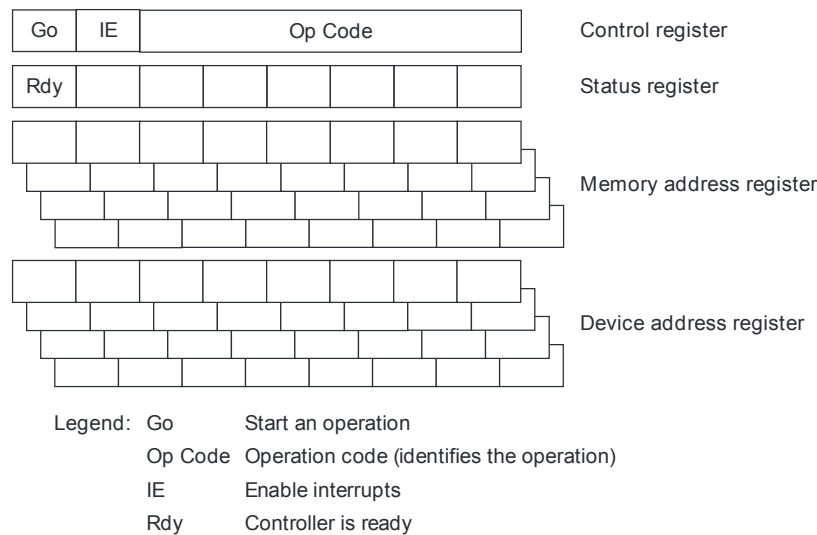


FIGURE 3.9 DMA registers.

Communicating with a PIO device is simple, though a bit laborious. For instance, to write one byte to the simulator's terminal device, you perform the following operations:

1. Store the byte in the *write register*.
2. Set the GoW bit in the *control register*.
3. Wait for RdyW bit (in the *status register*) to be set.

Rather than repeatedly checking the status register to determine if the RdyW bit has been set, you can request that an interrupt occur when it is set (indicating that the byte has been written). To do this, you set the IEW bit in the control register at the same time you set the GoW bit.

Communicating with a DMA device is a bit less laborious: the controller does the work. For example, to write the contents of a buffer to the disk you do the following:

1. Set the disk address in the *device address register*.
2. Set the buffer address in the *memory address register*.
3. Set the Op Code (Write), and the Go and IE bits in the *control register*.

The operation is complete when the IE bit is set, which it will be after the interrupt.

Most operating systems encapsulate device-specific details like those outlined above in collections of modules known as *device drivers*. These provide a standard interface to the rest of the operating system, which can then treat I/O in a device-independent manner.

If an operating system were written in C++, a device driver would be a class, with an instance for each device. We might have a base class for disks, with different types of disks being derived classes. The base class would have declarations of virtual functions for a standard disk interface, while the derived class would supply the actual definitions. A class declaration for a simple disk driver supporting a strictly synchronous interface might look like:

```
class disk {
public:
    virtual status_t read(request_t);
    virtual status_t write(request_t);
    virtual status_t interrupt();
};
```

Within the operating system, threads (perhaps user threads in the midst of system calls) would call *read* and *write* to perform the indicated operation, waiting until completion. When an interrupt occurs, the driver's interrupt method is called in the interrupt context and, when warranted, releases a thread waiting within *read* or *write*.

Even in operating systems providing a strictly synchronous I/O interface to users, such as Sixth-Edition Unix, the internal driver interface is often asynchronous (we discuss this in Chapter 4). Thus a more realistic class declaration for a driver might look like:

```
class disk {
public:
    virtual handle_t start_read(request_t);
    virtual handle_t start_write(request_t);
    virtual status_t wait(handle_t);
    virtual status_t interrupt();
};
```

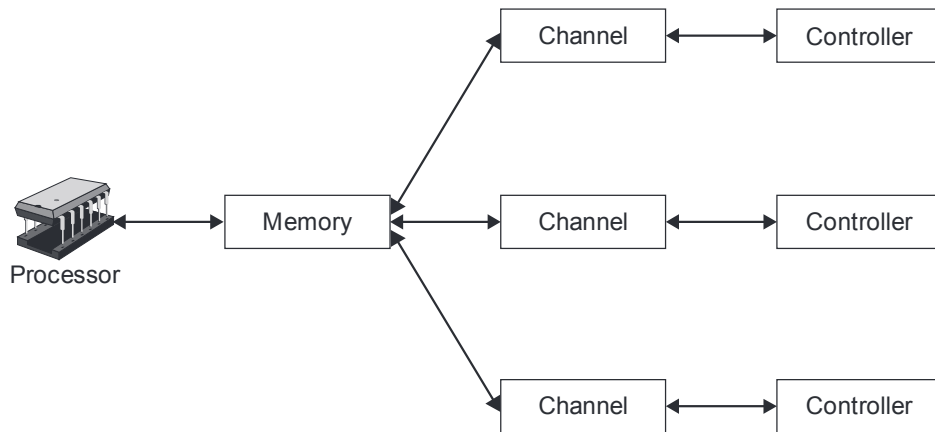


FIGURE 3.10 I/O with channels.

In this version, the *start_read* and *start_write* methods return a handle identifying the operation that has started. A thread can, at some later point, call *wait* with the handle and wait until that operation has completed. Note that multiple threads might call *wait* with the same handle if all must wait for the same operation (for example, if all need the same block from a file).

A more sophisticated approach to I/O used in what are commonly called “mainframe” computers is to employ specialized I/O processors to handle much of the I/O work. This is particularly important in large data-processing applications where I/O costs dominate computation costs. These I/O processors are traditionally called *channels* and execute programs in primary storage called *channel programs* (see Figure 3.10).

Storage allocation is a very important concern in operating systems. Whenever a thread is created, its stacks, control block, and other data structures must be allocated; whenever a thread terminates, these data structures must be freed. Since there are numerous other such dynamic data structures, both inside the operating system and within user applications, this allocation and liberation of storage must be done as quickly as possible.

The classic reference for most of this material is Donald Knuth’s *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. In this section we summarize Knuth’s results and discuss some additional operating-system concerns.

3.3 DYNAMIC STORAGE ALLOCATION

3.3.1 BEST-FIT AND FIRST-FIT ALGORITHMS

We start with the general problem of maintaining a pool of available memory and allocating variable-size quantities from it. Following Knuth (this is his example), consider the memory pool in Figure 3.11 that has two blocks of free memory, one 1300 bytes long and the other 1200 bytes long. We’ll try two different algorithms to process series of allocation requests. The first is called *first fit* — an allocation request is taken from the first area of memory that is large enough to satisfy the request. The second is called *best fit* — the request is taken from the smallest area of memory that is large enough to satisfy the request.

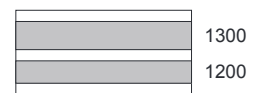


FIGURE 3.11 Pool of free storage.

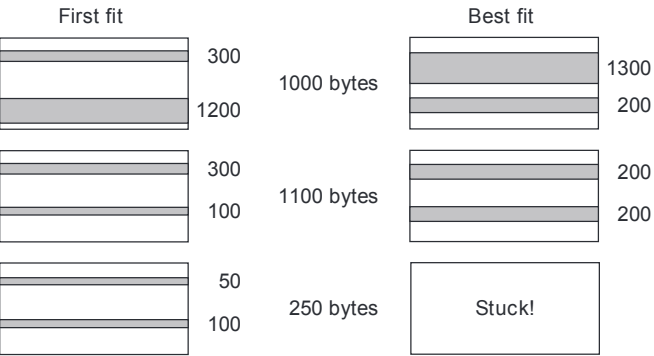


FIGURE 3.12 Allocations using first fit and best fit.

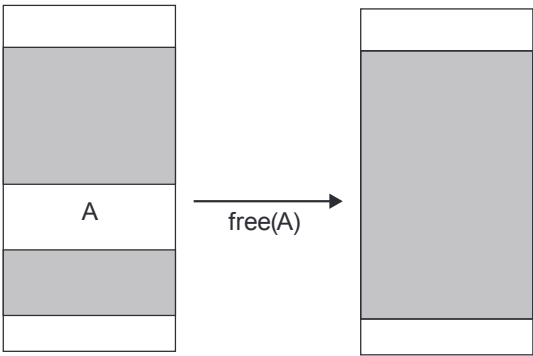


FIGURE 3.13 Liberating storage.

On the principle that whatever requires the most work must be best, one might think that best fit is the algorithm of choice. However, Figure 3.12 illustrates a case in which first fit behaves better than best fit. We first allocate 1000 bytes. Under the first-fit approach (on the left), this allocation is taken from the topmost region of free memory, leaving behind a region of 300 bytes of still unallocated memory. With the best-fit approach (right), this allocation is taken from the bottommost region of free memory, leaving behind a region of 200 bytes of still unallocated memory. The next allocation is for 1100 bytes. Under first fit, we now have two regions of 300 bytes and 100 bytes. Under best fit, we have two regions of 200 bytes. Finally, there is an allocation of 250 bytes. First fit leaves behind two regions of 50 bytes and 100 bytes, but best fit cannot handle the allocation — neither remaining region is large enough.

Clearly one can come up with examples in which best fit performs better. However, Knuth’s simulation studies have shown that, on the average, first fit works best. Intuitively, the reason for this is that best fit tends to leave behind a large number of regions of memory that are too small to be useful, as in Figure 3.12.

The liberation of storage is more difficult than its allocation, for the reason shown in Figure 3.13 (another example from Knuth). Here the shaded regions are unallocated memory. The region of storage A separating the two unallocated regions is about to be liberated. We’d like this to produce one large region of unallocated storage rather than three smaller adjacent regions. Thus the liberation algorithm must be able to determine if adjacent storage regions are allocated or free. An algorithm could simply walk through the list of free storage to determine if the adjacent areas are free, but a much cheaper approach is to tag the boundaries of storage regions to indicate whether they are allocated or free. Knuth calls this the *boundary-tag* method and provides an algorithm for it.

The shortcomings of best fit illustrate a common issue in storage management: *fragmentation*. What we saw here is called *external fragmentation*, in which we end up with lots of small areas of storage that collectively are sizeable, but individually are too small to be of use. In the following sections we encounter *internal fragmentation*, in which storage is wasted because more must be allocated than is needed.

3.3.2 BUDDY SYSTEM

The buddy system is a simple dynamic storage allocation scheme that works surprisingly well. Storage is maintained in blocks whose sizes are powers of two. Requests are rounded up to the smallest such power greater than the request size. If a block of that size is free, it’s taken; otherwise, the smallest free block larger than the desired size is found and split in half — the two halves are called *buddies*. If the size of either buddy is what’s needed, one of them is allocated (the other remains free). Otherwise one of the buddies is split in half. This splitting continues until the appropriate size is reached.

Liberating a block is now easy: if the block's buddy is free, you join the block being liberated with its buddy, forming a larger free block. If this block's buddy is free, you join the two of them, and so forth until the largest free block possible is formed.

One bit in each block, say in its first or last byte, is used as a tag to indicate whether the block is free. Determining the address of a block's buddy is simple. If the block is of size 2^k , then the rightmost $k-1$ bits of its address are zeros. The next bit (to the left) is zero in one buddy and one in the other; so you simply complement that bit to get the buddy's address.

Turning this into a data structure and algorithm is pretty straightforward. We maintain an array of doubly linked lists of free blocks, one list for each power of two. Free blocks are linked together using fields within them (which certainly aren't being used for anything else). See (Knuth 1968) for details.

3.3.3 SLAB ALLOCATION

Many kinds of objects in operating-system kernels are allocated and liberated frequently. Allocation involves finding an appropriate-sized area of storage, then initializing it, i.e., initializing various pointers or setting up synchronization data structures. Liberation involves tearing down the data structures and freeing the storage. If, for example, the storage space is allocated using the buddy system and the size of the objects is not a power of two, there's a certain amount of loss due to internal fragmentation. Further, a fair amount of time overhead may arise from initialization and tearing down.

Slab allocation (Bonwick 1994) is a technique in which a separate cache is set up for each type of object to be managed. Contiguous sets of pages called *slabs* are allocated to hold objects. Whenever a slab is allocated, a constructor is called to initialize all the objects it holds. Then as objects are allocated, they are taken from the set of existing slabs in the cache. When objects are freed, they are simply marked as such and made available for reallocation without freeing their storage or tearing them down. Thus new objects can be allocated cheaply. When storage is given back to the kernel (either because there are too many free objects or because the kernel requests storage due to a system-wide storage shortage), entire slabs are returned and each object in them is appropriately torn down.

A further benefit of slab allocation is "cache coloring": if all instances of an object are aligned identically, then all occupy the same cache lines and thus only one (or only a few, depending on the cache) can be in the cache at once. However, we can make successive slabs start the run of objects at different offsets from the beginning of the slab. Runs starting at such different offsets are said to have different colors; thus different-colored objects can be in the cache at the same time.

Linking and loading are apparently simple concepts that involve amazing complexity. They entail piecing together programs from their various components and adjusting addresses used in the resulting program so they refer to what the programmer intended.

We discuss the basic concepts of linking and loading here; for details on specific systems see <http://msdn.microsoft.com/en-us/library/ms809762.aspx> for Microsoft's Portable Execution (PE) format and <http://dlc.sun.com/pdf/816-1386/816-1386.pdf> for Sun's version of executable and linking format (ELF). Both are supported on Linux.

3.4 LINKING AND LOADING

3.4.1 STATIC LINKING AND LOADING

Let's examine what it takes to convert a program in C into code that is ready for execution on a computer. Consider the following simple program:

```
int main(int argc, char *[]) {
    return(argc);
}
```

Compiling it into x86 assembler language might produce:

```
main:
    pushl   %ebp                ; Push frame pointer.
    movl    %esp, %ebp          ; Set frame pointer to point to new
                                ; frame.
    movl    8(%ebp), %eax        ; Put argc into return register (eax).
    movl    %ebp, %esp          ; Restore stack pointer.
    popl    %ebp                ; Pop stack into frame pointer.
    ret                                ; Return: pop end of stack into eip.
```

If we run this code through an assembler and load it into our computer's memory starting at address 1000, say, we can then call it (with the stack pointer set correctly and the appropriate arguments on the stack), and it should do what was intended: return the value of *argc*. What's important is that its only reference to data was based on the value of register *ebp*, the frame pointer.

But now consider the following program:

```
int X=6;
int *aX = &X;

int main( ) {
    void subr(int);
    int y=X;
    subr(y);
    return(0);
}

void subr(int i) {
    printf("i = %d\n", i);
}
```

We don't need to look at the assembler code to see what's different: the machine code produced for it can't simply be copied to an arbitrary location in our computer's memory and executed. The location identified by the name *aX* should contain the address of the location containing *X*. But since the address of *X* is not known until the program is copied into memory, neither the compiler nor the assembler can initialize *aX* correctly. Similarly, the addresses of *subr* and *printf* are not known until the program is copied into memory — again, neither the compiler nor the assembler would know what addresses to use.

We might try solving this problem by always loading *main* at some standard location, such as 0. But suppose *subr* and the declaration for *X* are contained in a separate file. Sometime after it is compiled and assembled, the resulting code is combined with that of *main*. But it's only then, when the code is combined, that the locations of *X* and *subr* become known. Then, after we've determined the locations of *X* and *subr*, we must modify any references to them to contain their addresses. And, of course, we have to find the code for *printf* and include that in our program.

Modules such as *main* and *subr* that require modification of internal references to memory are said to be *relocatable*; the act of modifying such a module so that these references

refer to the desired target is called *relocation*. The program that performs relocation is called a *loader*.

Let's now look at a slightly restructured version of our program and work things out in detail.

File *main.c*:

```
extern int X;
int *aX = &X;

int main( ) {
    void subr(int);
    int y = *aX;
    subr(y);
    return(0);
}
```

File *subr.c*:

```
#include <stdio.h>
int X;

void subr(int i) {
    printf("i = %d\n", i);
}
```

We might compile the program on Unix with the command:

```
cc -o prog main.c subr.c
```

This causes the contents of the two files to be compiled directly into machine code and placed in files *main.o* and *subr.o* (the o stands for “object file”). These files are then passed to the *ld* program (the Unix linker and loader), which combines them into an executable program that it stores in the file *prog*. Of course, it also needs the code for *printf*, which is found in a library file whose name is known to *ld*. The resulting file *prog* can be loaded into memory through the actions of one of the *exec* system calls; thus *prog* must indicate what goes into text, what goes into data, and how much space to reserve in BSS (see Section 1.3.2).

How does *ld* determine what is to be done? The file *main.c* contains references to two items defined elsewhere: *X* and *subr*. We had to declare *X* explicitly as defined elsewhere, while the fact that we've provided no definition for *subr* is an implicit indication that *subr* is defined elsewhere. As *ld* produces the executable *prog*, it must copy the code from *main.o* into it, adjusting all references to *X* and *subr* so that they refer to the ultimate locations of these things. Instructions for doing this are provided in *main.o*. What these entail is first an indication that *ld* must find definitions for *X* and *subr*. Then, once their locations are determined, the instructions must indicate which locations within *main.o* must be modified when copied to *prog*. Thus *main.o* must contain a list of external symbols, along with their type, the machine code resulting from compiling *main.c*, and instructions for updating this code.

To see what's going on, look at the result of compiling *main.c* into Intel x86 assembler code.

Offset	Op	Arguments
0:	.data	; Initialized data follows.
0:	.globl	aX ; aX is global: it may be used by ; others.
0:aX:		
0:	.long	X
4:		
0:	.text	; Offset restarts; what follows ; is text (read-only code).
0:	.globl	main
0: main:		
0:	pushl	%ebp ; Save the frame pointer.
1:	movl	%esp,%ebp ; Set it to point to current ; frame.
3:	subl	\$4,%esp ; Make space for local variable y ; on stack.
6:	movl	aX,%eax ; Put contents of X into register ; eax.
11:	movl	(%eax),%eax ; Put contents of memory pointed ; to by X into %eax.
13:	movl	%eax,-4(%ebp) ; Store *aX into y.
16:	pushl	-4(%ebp) ; Push y onto stack.
19:	call	subr
24:	addl	\$4,%esp ; Remove y from stack.
27:	movl	\$0,%eax ; Set return value to 0.
31:	movl	%ebp, %esp ; Restore stack pointer.
33:	popl	%ebp ; Pop frame pointer.
35:	ret	
36:		

The code starts with an explicit directive to the assembler that what follows should be placed in the initialized data section. The *.globl* directive means that the following symbol (*aX*) is defined here, but may be used by other modules. Then we reserve four bytes of memory (a *long*), refer to it as *aX*, and indicate that its initial value is *X*, which is not further defined in this file. Next is a directive indicating that what follows should be placed in the text section, and that *main* is also a global symbol and can be used by others.

There are three locations requiring relocation. The reference to *X* in the data section must be replaced with the address where *X* is actually stored. The *movl* instruction at offset 6 in the text area contains a reference to *aX*. This must be replaced with the address of *aX*; the instruction occupies five bytes and this address goes into the upper four bytes, in other words, into the four bytes starting at offset 7. Similarly, the *call* instruction at offset 19 contains a reference to *subr*. This must be replaced with *subr*'s address: the instruction occupies five bytes and the address goes into the four bytes starting at offset 20. However, a peculiarity of the Intel x86 architecture is that such *call* instructions refer to their targets via "PC-relative" addressing. Thus what is stored at offset 20 is not the absolute address of *subr*, but its signed offset relative to the location of the instruction following the *call* (offset 24).

The result of compiling *subr.c* into Intel x86 assembler code is:

Offset	Op	Arguments	
0:	.data		; What follows is initialized data
0:	printfarg:		
0:	.string	"i = %d\n"	
8:			
0:	.comm	X, 4	; 4 bytes in BSS is required for ; global X.
4:			
0:	.text		; Offset restarts; what follows is ; text (read-only code).
0:	.globl	subr	
0:	subr:		
0:	pushl	%ebp	; Save the frame pointer.
1:	movl	%esp, %ebp	; Set frame pointer to point to ; current frame.
3:	pushl	8(%ebp)	; Push i onto stack.
6:	pushl	\$printfarg	; Push address of string onto stack.
11:	call	printf	
16:	addl	\$8, %esp	; Pop arguments from stack.
19:	movl	%ebp, %esp	; Restore stack pointer.
21:	popl	%ebp	; Pop frame pointer
23:	ret		
24:			

This file specifies that one item goes into initialized data: *printfarg*, which is a constant used locally. Then *X* is defined as being *common*,³ which means that it requires space in BSS. Within the text are the global symbol *subr* and a reference to the undefined symbol *printf*.

The object files produced by the assembler (or directly from the C code by the compiler) describe each of the three sections' data, BSS, and text. Along with each section is a list of global symbols, undefined symbols, and instructions for relocation. These instructions indicate which locations within the section must be modified and which symbol's value is used to modify the location. A symbol's value is the address that is ultimately determined for it; typically this address is added to the location being modified.

Here there are two locations requiring relocation. The *pushl* instruction at offset 6 contains a reference to *printfarg*, which must be replaced with the actual address. The instruction occupies five bytes and the address goes into the four bytes starting at offset 7. The *call* instruction at offset 11 contains a reference to *printf* that also must be replaced with the PC-relative address of *printf*'s final location. Again, the instruction occupies five bytes, and thus the address goes into the four bytes starting at offset 12.

So, in our example, *main.o*, the object file for *main*, would contain the following information.

³ The term *common* originated in Fortran and meant essentially the same thing as global data in C. Initialized common was possible, but required additional syntax.

```

Data:
  Size:      4
  Global:    aX, offset 0
  Undefined: X
  Relocation: offset 0, size 4, value: address ofX
  Contents:  0x00000000

BSS:
  Size:      0

Text:
  Size:      36
  Global:    main, offset 0
  Undefined: subr
  Relocation: offset 7, size 4, value: address of aX
               offset 20, size 4, value: PC-relative address of
               subr
  Contents:  [machine instructions]

```

And *subr.o* would contain:

```

Data:
  Size:      8
  Contents:  "i = %d\n"

BSS:
  Size:      4
  Global:    X, offset 0

Text:
  Size:      44
  Global:    subr, offset 0
  Undefined: printf
  Relocation: offset 7, size 4, value: address of printfarg
               offset 12, size 4, value: PC-relative address of
               printf
  Contents:  [machine instructions]

```

Among the duties of *ld*, the linker-loader, is tracking down all the unresolved references. After it processes *main.o*, the unresolved references are *X* and *subr*. Its next input, *subr.o*, contains definitions for both, but has its own unresolved reference: *printf*. This and any other unresolved references are looked for in a list of libraries supplied to *ld*. We haven't supplied any libraries to it explicitly (we might, for example, have it look in the *pthread*s library if we were compiling a multithreaded program), but it always looks in the *standard C* library for anything not yet resolved. Here it finds *printf*, which, it turns out, contains its own unresolved reference: *write*. This is also in the standard C library, which is searched repeatedly for any unresolved references stemming from modules obtained from itself.

Assume the object code for *printf* is:

```
Data:
  Size:      1024
  Global:    StandardFiles
  Contents:  ...

BSS:
  Size:      256

Text:
  Size:      12000
  Global:    printf, offset 100
  ...
  Undefined: write
  Relocation: offset 211, value: address of StandardFiles
              offset 723, value: PC-relative address of write
  Contents:  [machine instructions]
```

Assume that the object code for *write* is:

```
Data:
  Size:      0

BSS:
  Size:      4
  Global:    errno, offset 0

Text:
  Size:      16
  Contents:  [machine instructions]
```

In addition, every C program contains a startup routine that is called first and then calls *main*. On return from *main*, this routine calls *exit*. Assume that its object code is:

```
Data:
  Size:      0

BSS:
  Size:      0

Text:
  Size:      36
  Undefined: main
  Relocation: offset 21, value main
  Contents:  [machine instructions]
```

The output of *ld* is an executable file, which is like an object file except that all references are fully resolved and no relocation is necessary. For our example, the executable file is *prog*. Based on its component object files, *ld* might set things up as follows:

Region	Symbol	Value
Text	main	4096
	subr	4132
	printf	4156
	write	16156
	startup	16172
Data	aX	16384
	printfargs	16388
	StandardFiles	16396
BSS	X	17420
	errno	17680

Note that *main* does not start at location 0: the first page is typically made inaccessible so that references to null pointers will fail. In order to make the text region read-only and the other regions read-write, the data region starts on the first (4096-byte) page boundary following the end of the text.

3.4.2 SHARED LIBRARIES

A drawback of the format of the executable files such as *prog* above is that they must contain everything needed for execution. This results in a fair amount of duplicate code. For example, almost every C program calls *printf*, and thus must contain a copy of it. All these copies of *printf* waste a huge amount of disk space. Furthermore, when each program is loaded into primary memory, the multiple copies of *printf* waste that memory too. What is needed is a means for programs to share a single copy of *printf* (and other routines like it).

Suppose we want just a single copy of *printf* on each computer. What is required to share it? If *printf* required no relocation, then sharing would be simple: we'd simply copy it from disk to primary memory when required, and every program using it could simply call it as needed. But *printf* does require relocation, so things aren't quite so simple.

One approach to sharing might be to have a single copy of *printf* on disk, shared by all programs. When a program using *printf* is loaded into memory, it then copies *printf* into its address space, performing the necessary relocation. This lets us reduce the disk wastage — there's just one disk copy — but every running program still has a private copy in memory.

A way to get around this problem is have all processes that use *printf* have it in the same location in their address spaces. Thus, for example, everyone might agree that if they use *printf*, they will put it at location 100,000 in their address spaces. We would “prerelocate” *printf* so that it works as long as it is placed in memory at location 100,000. This would let us have just one copy of *printf* in memory as well as on disk. Of course, if for some reason a program has something else at location 100,000, this won't work.

This technique of prerelocation has been used in some Unix systems and a form of it is currently used in Windows (see Exercise 9). If for some reason prerelocation does not work (say, because the intended location in the address space is taken), then the code is “rerelocated” to fit someplace else.

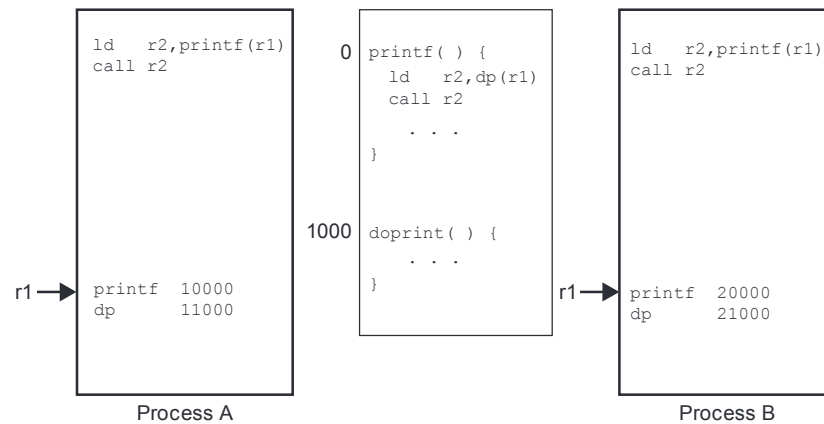


FIGURE 3.14 Position-independent code.

The final approach is to make relocation unnecessary, producing code that can be placed anywhere in memory without requiring modification. Such code is known as *position-independent code* or *PIC*. Some magic is required to make this work; the magic takes the form of indirection. PIC code does not contain any addresses; instead, it refers to process-specific tables that in turn contain addresses.

Figure 3.14 shows an example of position-independent code. Processes A and B are sharing the library containing *printf* (note that *printf* contains a call to another shared routine, *dp*), though they have it mapped into different locations. Each process maintains a private table, pointed to by register `r1`. In the table are the addresses of shared routines, as mapped into the process. Thus, rather than call a routine directly (via an address embedded in the code), a position-independent call is made: the address of the desired routine is stored at some fixed offset within the table. The contents of the table at this offset are loaded into register `r2`, and then the call is made via `r2`.

Shared libraries, whether implemented using preresolution or with position-independent code, are used extensively in many modern systems. In Windows they are known as *dynamic-link libraries* (DLLs) and in Unix systems they are known as *shared objects*. An advantage of using DLLs and shared objects is that they need not be loaded when a program starts up, but can be loaded only if needed, thus reducing the start-up time of the program. A disadvantage is that their use can add to the complexity of a program, both through dependencies (DLL A uses code from DLL B, which uses code from DLL C, etc.) and from versioning (there's a new version of DLL B: is it compatible with my versions of DLLs A and C?).

We conclude this section with a detailed description of how shared objects are implemented on the x86 in ELF (executable and linking format), which is used on most Unix systems, including Linux, Solaris, FreeBSD, NetBSD, and OpenBSD, but not MacOS X. It is not used on Windows.

When a program is invoked via the *exec* system call, the code that is first given control is *ld.so* — the runtime linker. It does some initial set up of linkages, as explained shortly, and then calls the actual program code. It may be called upon later to do some further loading and linking, as explained below.

ELF requires three data structures for each dynamic executable (i.e., the program binary loaded by the *exec* system call) and shared object: the *procedure-linkage table*, the *global-offset table*, and the *dynamic structure*. To simplify discussion, we refer to dynamic executables and shared objects as *modules*. The procedure-linkage table contains the code that's actually called when control is to be transferred to an externally defined routine. It is shared by all processes using the associated executable or object, and uses data in the global-object table to link the

- Procedure-linkage table
 - shared, read-only executable code
 - essentially stubs for calling subroutines
- Global-offset table
 - private, read-write data
 - relocated dynamically for each process
- Dynamic structure
 - shared, read-only data
 - contains relocation info and symbol table

FIGURE 3.15 Data structures used by ELF to support shared objects.

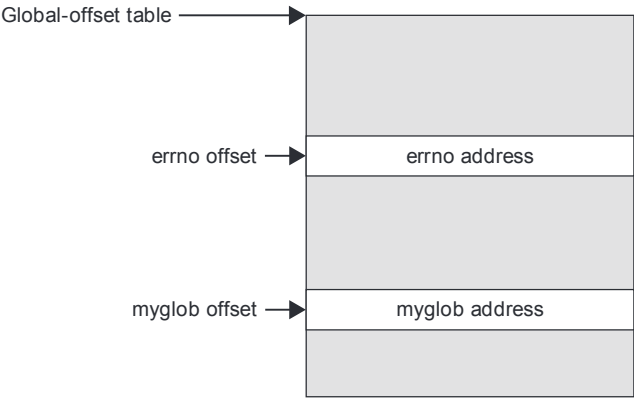


FIGURE 3.16 The global-offset table (GOT).

caller to the called program. Each process has its own private copy of each global-object table. It contains the relocated addresses of all externally defined symbols. Finally, the dynamic structure contains much information about each module. This is summarized in Figure 3.15.

To establish position-independent references to global variables, the compiler produces, for each module, a global-offset table (GOT). Modules refer to global variables indirectly by looking up their addresses in the table: a register contains the address of the table and modules refer to entries via their offsets. When the module is loaded into memory, *ld.so* is responsible for putting into it the actual addresses of all the needed global variables. When the program starts up, *ld.so* does this for the dynamic executable. As shared objects are loaded in (how and when this is done is covered soon), *ld.so* sets up their GOTs. Figure 3.16 shows an example in which *errno* and *myglob* are global variables whose locations are stored in global-offset-table entries.

Dealing with references to external procedures is considerably more complicated than dealing with references to external data. Figure 3.17 shows the procedure linkage table, global offset table, and relocation information for a module that contains references to external procedures *name1* and *name2*. Let's follow a call to procedure *name1*. The general idea is before the first call to *name1*, the actual address of the *name1* procedure is not recorded in the global-offset table. Instead, the first call to *name1* invokes *ld.so*, which is passed parameters indicating what is really wanted. It then finds *name1* and updates the global-offset table so that on subsequent calls *name1* is invoked more directly. To make this happen, references from the module to *name1* are statically linked to entry *.PLT1* in the procedure-linkage table. This entry contains an unconditional jump to the address contained in the *name1* offset of the global-offset table (pointed to by register *ebx*). Initially this address is of the instruction following the jump instruction, which contains code that pushes onto the stack the offset of the *name1* entry in the relocation table.

The next instruction is an unconditional jump to the beginning of the procedure-linkage table, entry *.PLT0*. Here there's code that pushes onto the stack the contents of the second 32-bit word of the global-offset table, which contains a value identifying this module.

The following instruction is an unconditional jump to the address in the third word of the global-offset table, which is conveniently the address of *ld.so*. Thus control finally passes to *ld.so*, which looks back on the stack and determines which module has called it and what that module really wants to call. It figures this out using the module-identification word and the relocation table entry, which contains the offset of the *name1* entry in the global-offset table (which is what must be updated) and the index of *name1* in the symbol table (so it knows the name of what it must locate).

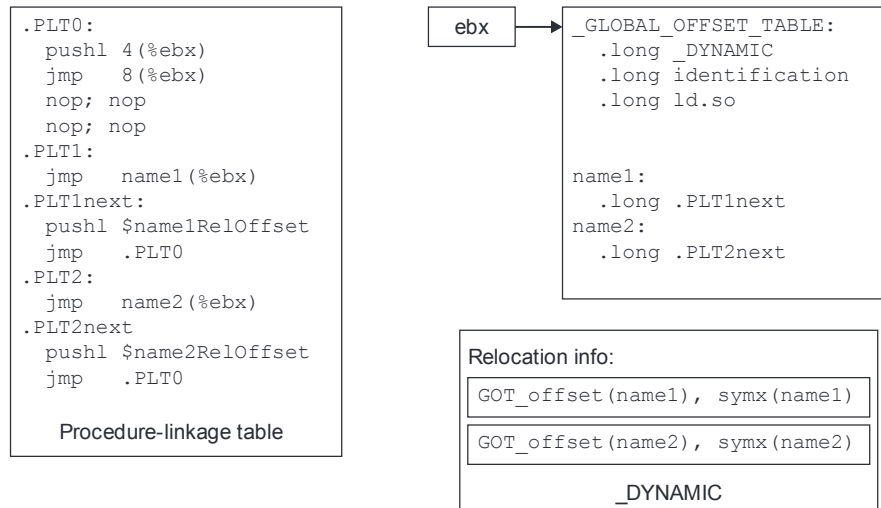


FIGURE 3.17 The procedure-linkage table, global-offset table, and dynamic structure before the first call to *name1*.

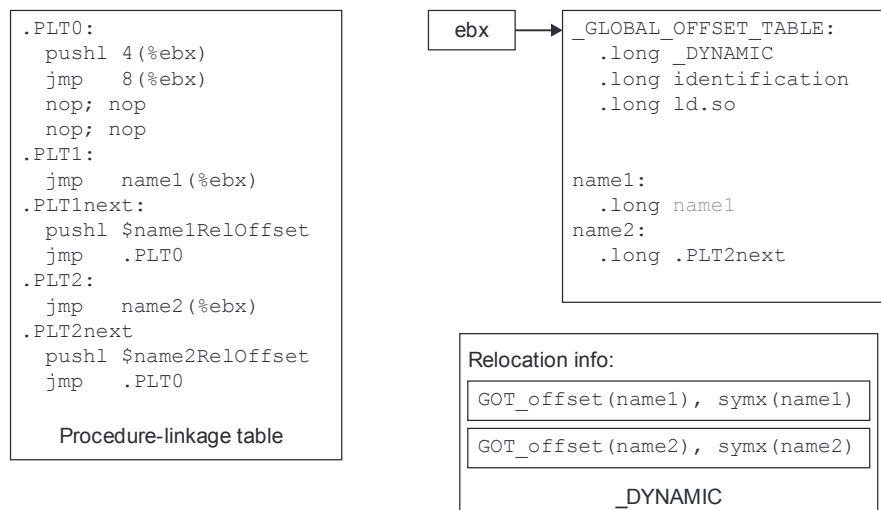


FIGURE 3.18 The procedure-linkage table, global-offset table, and dynamic structure after the first call to *name1*.

Finally, *ld.so* writes the actual address of the *name1* procedure into the *name1* entry of the global-offset table and, after unwinding the stack a bit, passes control to *name1*. On subsequent calls by the module to *name1*, since the global-offset table now contains *name1*'s address, control goes to it more directly, without invoking *ld.so*. The updated tables are shown in Figure 3.18.

We finish this chapter with a discussion of booting an operating system. The term “boot” is probably derived from the idiomatic expression “to pull yourself up by your bootstraps,” which means to get out of a difficult situation solely by your own efforts. The difficult situation here is that a computer must load its operating system into memory, and this task requires having



FIGURE 3.19 A DEC PDP-8.⁴ Note the row of switches used to select memory addresses and insert values into memory. (Courtesy of David Gesswein.)

an operating system in memory. To get around this circularity, we first get a really tiny operating system into memory — the bootstrap loader — and it reads the real operating system into memory. (Or perhaps it reads a somewhat larger bootstrap loader into memory, which then fetches the real operating system.) But we still wonder: how do we get the first bootstrap loader into memory?

Early computers employed a conceptually easy approach. For example, the DEC PDP-8 (Figure 3.19) had a set of console switches through which one would manually put into memory a simple bootstrap loader, which would then read a more complicated program (perhaps an operating system) from paper tape. The code below is one such bootstrap loader that one would “toggle in” by hand:

```
07756 6032 KCC
07757 6031 KSF
07760 5357 JMP .-1
07761 6036 KRB
07762 7106 CLL RTL
07763 7006 RTL
07764 7510 SPA
07765 5357 JMP 7757
07766 7006 RTL
07767 6031 KSF
07770 5367 JMP .-1
07771 6034 KRS
07772 7420 SNL
07773 3776 DCA I 7776
07774 3376 DCA 7776
```

⁴The photo is from <http://www.pdp8.net/pdp8i/pdp8i.shtml>.

```

07775 5356 JMP 7756
07776 0000 AND 0
07777 5301 JMP 7701

```

The first column is the memory address (in octal), the second column is the code one would put into that location, and the third column gives the assembler instructions encoded.

Such an approach had obvious limitations. DEC, with its 1978 VAX-11 series, replaced the toggle switches with a separate console subsystem consisting of a specialized processor, read-only memory, a floppy-disk drive, and a hard-copy terminal. Rather than toggle in code to be deposited in the main processor's memory, one could type in commands to have the code deposited. This by itself was no improvement, but one could also type in commands to read a bootstrap program from the floppy disk into the main processor's memory. Furthermore, the computer could be configured to do this automatically when it powered up. How did one boot the specialized processor in the console subsystem? It was "hard-wired" always to run the code contained in its read-only memory when it started up.

The bootstrap loader loaded from the floppy disk would handle both the disk device and the on-disk file system (see Chapter 6) well enough to follow a directory path and read a file containing the operating system. It loaded the operating system into memory, then passed control to it. To do all this, the bootstrap code had to have the correct driver for the disk containing the file system that contained the operating system, and it had to know how that disk was set up. The latter involves both what sort of file system to expect and how the disk is partitioned — a single disk might hold multiple different file-system instances, each in a separate region of the disk.

The operating system, once loaded, had to be able to handle whatever devices were present on the system as well as however the disk was partitioned. Early Unix systems did this by having all needed device drivers statically linked in them, as well as by having the disk-partitioning information statically configured in the device drivers. Adding a new device, replacing a device, modifying disk-partitioning information, and so forth all required compiling (not just relinking) a new version of the operating system.

Later versions of Unix improved things a bit by containing all possible device drivers, then probing at boot time to determine which were actually needed. Disk-partition tables, rather than being static, were put in the first sector of each disk and read in at boot time, allowing the system to configure itself. Still later versions of Unix allowed device drivers to be dynamically loaded into a running system, thus easing the logistics of managing systems with disparate devices.

IBM introduced the IBM PC in 1981, establishing both a model and a marketplace in which a computer might include devices from any of a large number of manufacturers. The operating system (MS-DOS, from Microsoft) was distributed in binary form only. It had to be booted from a variety of devices and, once running, had to handle a large variety of devices. Not only was source code unavailable, but the typical user would not have known what to do with it. So a different approach was devised.

Included with the IBM PC is the basic input/output system (BIOS), consisting of code stored in read-only memory (ROM) and various settings stored in non-volatile memory (such as CMOS). It provides three primary functions:

- power-on self test (POST)
- load and transfer control to the boot program
- provide drivers for all devices

The provider of the computer system supplies a BIOS chip residing on the “motherboard” containing everything necessary to perform the three functions. Additional BIOS functions might reside on chips on add-on boards, providing access to additional devices.

The BIOS ROM is mapped into the last 64K of the first megabyte of address space (starting at location 0xf0000). When the system is powered on, the processor starts executing instructions at location 0xffff0 — the last sixteen bytes of this mapped region. At this location is a branch instruction that transfers control to the beginning of the BIOS code. The first thing the code does is the power-on self test, during which it initializes hardware, checks for problems, and computes the amount of primary storage.

The next step is to find a boot device. This is probably a hard disk, but could be a floppy or diskette. The list of possibilities and the order to search are in the settings stored in non-volatile RAM.

Once the boot device is found, the master boot record (MBR) is loaded from the first sector of a floppy or diskette, or from cylinder 0, head 0, sector 1 of a hard disk (see Chapter 6 for an explanation of these terms). The MBR (Figure 3.20) is 512 bytes long and contains:

- a “magic number” identifying itself
- a partition table listing up to four regions of the disk (identifying one as the *active* or boot partition)
- executable boot program

The BIOS code transfers control to the boot program. What happens next depends, of course, on the boot program. In the original version (for MS-DOS), this program would find the one active partition, load the first sector from it (containing the *volume boot program*), and pass control to that program. This program would then load the operating system from that partition.

More recent boot programs allow more flexibility. For example, both lilo (Linux Loader) and grub (Grand Unified Boot Manager), allow one to choose from multiple systems to boot, so that, for example, one can choose between booting Linux or Windows. Lilo has the sector number of the kernel images included within its code and thus must be modified if a kernel image moves. Grub understands a number of file systems and can find the image given a path name.

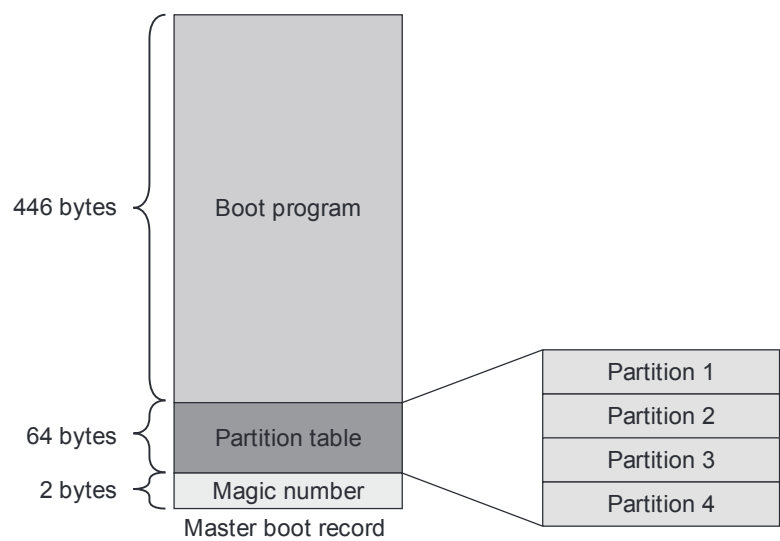


FIGURE 3.20 The master boot record, residing in the first sector of a bootable disk.

The next step is for the kernel to configure itself. In Linux, most of the kernel is stored in the file system in compressed form. Here are the steps Linux goes through.

1. Performed by uncompressed code written in assembler language: set up initial stack, clear BSS, uncompress kernel, then transfer control to it.
2. Performed by newly uncompressed code, also written in assembler language: set up initial page tables and turn on address translation (see Chapter 7). At this point, process 0, which handles some aspects of paging, is created.
3. Performed by newly uncompressed code written in C: the rest of the kernel is initialized and process 1, which is the ancestor of all other user processes, is created. The scheduler is invoked, which schedules process 1.

We still need to discuss the issue of devices. One of the jobs of BIOS is to determine which devices are present. Originally, it provided drivers for all devices and the operating system would call upon BIOS-provided code whenever it required services of a device driver. But, since such drivers occupied rather slow memory and provided minimal functionality, later systems copied the BIOS drivers to primary memory, and even later ones provided their own device drivers, working through a list of devices provided by BIOS. However, BIOS drivers are still used for booting, since the operating system must be read from a disk with the aid of some sort of driver.

BIOS was designed in the 1980s for x86 computers and is not readily extensible to other architectures. Since then, Sun introduced Open Firmware, which is now used by a number of systems that don't use the x86. It is based on the language Forth, whose code is compiled into byte code and interpreted directly.

Intel developed a replacement for BIOS called extensible firmware interface (EFI) that also uses byte code, known imaginatively as EFI byte code. EFI development continues under the aegis of the UEFI forum, which is a group of a number of interested companies, including Intel and Microsoft.

This chapter has covered a number of topics of use throughout the rest of the book. Understanding context switching is essential to understanding the management of processors, which we cover in Chapter 5. Our brief discussion of input/output architectures prepares us to discuss file systems in Chapter 6. Dynamic storage allocation comes into play in all parts of operating systems, but most importantly in memory management, which we cover in Chapter 7. We will not have direct contact with linking and loading again, but it is clearly important for running programs. Similarly, we will not discuss booting again, but it is clearly important.

3.6 CONCLUSIONS

1. Why does the x86 architecture require both a frame-pointer register and a stack-pointer register?
- *2. Recursion, in the context of programming languages, refers to a function that calls itself. For example, the following is a simple example (assume that it's called only with appropriate argument values):

```
int factorial(int n) {
    if (n == 1)
        return n;
```

3.7 EXERCISES

```

    else
        return n*factorial(n-1);
}

```

Tail recursion is a restriction of recursion in which the result of a recursive call is simply returned — nothing else may be done with the result. For example, here's a tail-recursive version of the factorial function:

```

int factorial(int n) {
    return f2(n, 1);
}

int f2(int a1, int a2) {
    if (a1 == 1)
        return a2;
    else
        return f2(a1-1, a1*a2);
}

```

- a. Why is tail recursion a useful concept? (Hint: consider memory use.)
 - b. Explain how tail recursion might be implemented so as actually to be so useful.
- * 3. Section 3.1.2 describes how to switch from one thread to another. The implicit assumption is that the thread being switched to in the *switch* routine has sometime earlier yielded the processor by calling *switch* itself. Suppose, however, that this thread is newly created and is being run for the first time. Thus when its creator calls *switch* to enter the new thread's context, the new thread should start execution as if its first routine had just been called. Show what the initial contents of its stack should be to make this happen. Assume an x86-like assembler language.
4. As discussed in Chapter 1, Unix's *fork* system call creates a new process that is a child of its parent, the calling process. The thread calling *fork* returns the process ID of the new process, but in the new process the newly created thread returns 0 to the user-mode code that called *fork*. Explain how the kernel stack of the new thread is set up to make this happen.
- * 5. As discussed in Chapter 5, many systems use the clock-interrupt handler to determine if the current thread's time slice is over and, if so, force the thread to yield the processor to another thread. If the system's architecture uses the current thread's kernel stack to handle interrupts, then the current thread's context was saved on its kernel stack when the interrupt occurred and, even though the system is running in the interrupt context, the interrupt handler might simply call *switch* to switch to a different thread (it's a bit more complicated than this since we must deal with the possibility of other interrupts; we take this up in Chapter 5).
- a. On architectures that have separate interrupt stacks, such as the DEC VAX-11, it does not work for an interrupt handler to call *switch* directly. Explain why not.
 - b. Explain what might be done in such an architecture for an interrupt handler to force a thread to call *switch* before normal execution resumes in the thread.
6. Assume we have a memory-mapped I/O architecture (Section 3.2) with device registers mapped into memory locations. In some situations it might be advantageous for a device driver, rather than enabling interrupts when it starts an I/O operation, to repeatedly check

the ready bit until the operation completes. Describe the circumstances in which this approach would be preferable to using interrupts.

7. Despite the discussion in Section 3.3.1, there are occasions in which it makes sense to use a best-fit storage allocation algorithm. Describe a situation in which best fit would be better than first fit.
- * 8. Pages 112 and 113 show the assembler code in *main.s* and *subr.s*. Describe what changes are made to the corresponding machine code (in *main.o* and *subr.o*) when these routines are processed by *ld*, forming *prog*. Be specific. In particular, don't say "the address of X goes in location Y," but say, for example, "the value 11346 goes in the four-byte field starting at offset 21 in *xyz.o*." Note that for the x86 instructions used in this example, an address used in an instruction appears in the four-byte field starting with the second byte of the instruction.
9. Microsoft chose not to use position-independent code in Windows. Instead, they use the prereslocation technique (Section 3.4.2). They presumably had good technical reasons for doing so. Can you explain what they might be?
10. Why doesn't *ld.so* (Section 3.4.2) fill in the procedure-linkage tables with the actual addresses of all the procedures when the program starts up?
- * 11. Many C and C++ compilers allow programmers to declare thread-specific data (Section 2.2.4) as follows.

```
__thread int X=6;
```

This declares X to be a thread-specific integer, initialized to 6. Somehow we must arrange so that each thread has its own private copy of it, initialized to 6. Doing this requires the cooperation of the compiler and the linker. Describe what must be done by both for this to work. Note that thread-specific data (TSD) items must be either global or static local variables. It makes no sense for them to be non-static local variables. You may assume that TSD items may be initialized only to values known at compile time (in particular, they may not be initialized to the results of function calls). Assume that only static linking is done — do not discuss the further work that must be done to handle dynamic linking. Note that a program might consist of a number of separately compiled modules.

12. Early versions of the Apple II computer had no operating system — many programs came with their own simple operating systems on self-booting floppy disks (see Section 1.2.4.1). Explain how such self-booting might have worked.

Bonwick, J. (1994). The Slab Allocator: An Object-Caching Kernel Memory Allocator. *Proceedings of the USENIX Summer 1994 Technical Conference*. Boston.

Knuth, D. E. (1968). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison Wesley (second edition 1973, third edition 1997).

SPARC International (1992). The SPARC Architecture Manual, Version 8. from <http://www.sparc.org/standards/V8.pdf>.

4.1	A Simple System	4.2	Rethinking Operating-System Structure
4.1.1	A Framework for Devices	4.2.1	Virtual Machines
4.1.2	Low-Level Kernel	4.2.1.1	Processor Virtualization
4.1.2.1	Terminals	4.2.1.2	I/O Virtualization
4.1.2.2	Network Communication	4.2.1.3	Paravirtualization
4.1.3	Processes and Threads	4.2.2	Microkernels
4.1.4	Storage Management	4.3	Conclusions
4.1.4.1	Managing Primary Storage	4.4	Exercises
4.1.4.2	File Systems	4.5	References

In the previous chapter we discussed many of the concepts used in operating systems. Now we begin to examine how operating systems are constructed: what goes into them, how the components interact, how the software is structured so as to make the whole understandable and supportable, and how performance concerns are factored in. We also introduce such key components as scheduling, virtual memory, and file systems here and cover them in detail in later chapters.

Our approach is to look at a simple hardware configuration and work through the design of an equally simple operating system for it. Many of the details are covered in depth in later chapters. Similarly, we examine more sophisticated operating-system features in later chapters as well.

As far as applications are concerned, the operating system is the computer. It provides processors, memory, files, networking, interaction devices such as display and mouse, and whatever else is needed. The challenge of operating-system design is to integrate these components into a consistent whole that provides a usable interface to a secure and efficient system.

Our primary concern is a “general-purpose” operating system, one that’s to run a variety of applications. Some of these applications are interactive, many of them utilize network communications, and all of them use and perhaps modify and create data on a file system. Examples of such general-purpose operating systems include Windows, Linux, FreeBSD, Solaris, and MacOS X (all but Windows are variants of Unix). Much of what we discuss applies to Chromium OS as well, since its kernel is Linux.

One way of looking at the design of such an operating system is to examine its functional components. Despite the marketing hype, Windows and most implementations of Unix are quite similar from a high-level point of view. All provide the notion of a process, which we can think of as a holder for the resources an application needs to run (Linux deviates from this a bit). All have threads, which, as we’ve seen, are an abstraction of the processor; they’re the active agents. All provide file systems, with similar operations for working with files. All

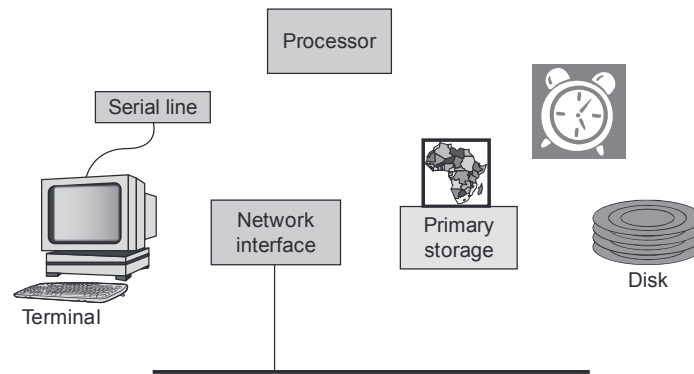


FIGURE 4.1
Hardware configuration
for simple OS.

implement roughly the same set of network protocols, with pretty much the same application interface. All provide for user interaction with a display, mouse, and keyboard. All provide access control to files that is based on a notion of file ownership and is controlled by the owner of files (this is known as *discretionary access control*, a concept we cover in Chapter 8).

In order to look at the basics of operating-system design, we sketch the design of a simple system that incorporates many of the major components of a modern general-purpose operating system. We work with a hardware configuration that dates to the early 1980s — we omit modern display technology so as to focus on the basic OS issues. Our system, as shown in Figure 4.1, is made up of a processor with memory and memory-management hardware (to support virtual memory) along with a disk drive, a clock, a serial line to which is connected a simple terminal device made up of a display and keyboard, and an Ethernet connection for communication. The I/O architecture is the memory-mapped approach discussed in Section 3.2.

Our operating system provides functionality similar to that of current Unix and Windows systems, but without support for bit-mapped displays and mice and with a simpler but generally less efficient design. It supports multiple multithreaded processes, each with its own address space, and has a file system layered on its disks. The execution of threads is multiplexed on the single processor by a simple, time-sliced scheduler. Users interact with the system via a terminal that provides no graphics, but displays the text sent to it (typically in 24 80-character rows) and sends to the processor whatever characters are typed on the keyboard (see Figure 4.2). Communication is provided over Ethernet using the TCP/IP family of protocols (we describe these protocols in Chapter 9). This system would have been state of the art (if not then some) in the 1970s, and rather commonplace by the early to mid-1980s.

4.1 A SIMPLE SYSTEM



FIGURE 4.2
A computer terminal
(~1980). (Courtesy of
Tom Doeppner.)

From an application program's point of view, our system has processes with threads, a file system, terminals (with keyboards), and a network connection. Before we look at how all this is implemented, let's go over these components in a bit more detail. We need to examine how applications use them and how this affects the design of the operating system.

The purpose of a process in our system is to hold both an address space and a group of threads that execute within that address space, as well as to keep track of files and devices in current use. One definition of an address space is the set of addresses that threads of the process can usefully reference. More practically, however, it is the contents of these addressable locations. As we saw in Chapter 1, the address spaces of typical Unix processes contain text, data, BSS, dynamic, and stack regions. Thus references to the text region obtain the application's executable code, references to the data region obtain its initialized variables, and so forth.

An important design issue is how the operating system initializes these address-space regions. A simple-minded approach would be to copy their contents from the file system to the process address space as part of the *exec* operation. However, particularly for the text region, this would be unnecessarily time- and space-consuming, since what is in this region is read-only and thus can, in principle, be shared by all processes executing code from the same file. It might seem that, since a process can modify the contents of its data region, the data region really must be copied into the address space. However, if portions of it are never modified, then perhaps a savings can be achieved by sharing those portions.

Rather than copy the executable file into the address space, a better approach is to *map* it into the address space. This is an operation we cover in detail in Chapter 7. Mapping lets the operating system tie the regions of the address space to the file system. Both the address space and files are divided into pieces, typically called pages. If, for example, a number of processes are executing the same program, at most one copy of that program's text pages is in memory at once. The text regions of all the processes running this program are set up, using hardware address-translation facilities, to share these pages. The data regions of all processes running the program initially refer to pages of memory containing a copy of the initial data region.

The sort of mapping employed here is called a *private mapping*: when a process modifies something in a data-region page for the first time, it is given a new, private page containing a copy of the initial page. Thus only those pages of memory that are modified are copied. The BSS and stack regions use a special form of private mapping: their pages are initialized, with zeros, only when processes access them for the first time.

This notion of mapping is used in more modern systems by allowing additional regions to be added to the address space that are set up by mapping other files into them. In addition to private mappings, *shared mappings* are used here: when a process modifies a page for the first time, a private copy is not made for the process; instead, the original page itself is modified. Thus all processes with a shared mapping of that page share the changes to it. Furthermore, these changes are written back to the original file.¹

Mapping files into the address space, both private and shared, is one way to perform input and output operations on files. We also need the more traditional approach using explicit system calls, such as Unix's *read* and *write*. These are necessary for interacting with devices that can't be mapped into the address space as files can, for instance receiving characters typed into the keyboard, or sending a message via a network connection. Such communication is strictly sequential: reading from the keyboard retrieves the next batch of characters; sending to the network connection transmits the next message.

¹ We've left out a number of details here. For example, how do we make room if more is mapped into the processes' address space than fits into memory? What happens if some processes have share-mapped a portion of a file into their address spaces while others have private-mapped the same portion into their address spaces? We cover all this, and more, in Chapter 6.

It also makes sense to use such explicit system calls when doing file I/O, even though file I/O can be done by mapping. For example, a program's input might be a sequence of commands. It makes life a lot easier to use the same code in the program to read this input from a file as to read it from the keyboard. A program that produces lines of text as output should be able to use the same code to write this output to file as to write it out to a network connection.

In the remainder of this section we focus on the design of our system. This design is relatively simple and straightforward; it makes clear what the components do and how they interact. We have to come to grips with a number of issues, including:

- What is the functionality of each of the components?
- What are the key data structures?
- How is the system broken up into modules?
- To what extent is the system extensible? That is, how easy is it to add new functionality or substitute different functionality?
- What parts run in the OS kernel in privileged mode? What parts run as library code in user applications? What parts run as separate applications?
- In which execution contexts do the various activities take place? In particular, what takes place in the context of user threads? What takes place in the context of system threads? What takes place in the interrupt context?

4.1.1 A FRAMEWORK FOR DEVICES

Before looking at how to handle individual devices, we need to devise a framework for handling them within our system. As we discussed in Chapter 3, device drivers contain the code that deals with device-level details, providing a standard interface to the rest of the system. But how do we connect the rest of the system to these device drivers? In particular, how do we make certain that, when a program reads from a keyboard, it is indeed reading from the keyboard device? How do we make certain that every time the system starts up, the file system is always associated with the same disk (or disks)? Taking this a step further, how does the system know what to do when we plug a new device into it, such as a flash drive?

In early Unix systems the kernel was statically linked to contain just the device drivers required, and each of them knew exactly the set of their devices that were physically connected to the system. There was no means for adding devices to a running system, or even for adding a device while the system is down without modifying the kernel image. If a system had two disk drives of the same sort, its kernel would have one driver for that sort of disk drive, configured to handle two instances of the drive. Each device was identified by a device number that was actually a pair: a *major device number*, identifying the driver, and a *minor device number* that was used, among other things, to identify the device among those handled by the driver. Special entries were created in the file system, usually in the `/dev` directory, to refer to devices. For our disk-drive example, `/dev/disk1` and `/dev/disk2` might be created, each marked as a *special file* — one that does not contain data, but instead refers to devices by their major and minor device numbers. If an application program opened `/dev/disk2`, the kernel would know to use the correct driver and the driver would now be able to identify which device, both being indicated by the device number contained in the special file.

The data structures required to support this driver framework were simple. There was a statically allocated array in the kernel called *cdevsw* (character² device switch) that was indexed by the major device number. Each entry in the array contained the addresses of the entry points of the indicated driver. (See Figure 4.3.) Each driver maintained its own data structures for identifying the device from the minor device number and for representing its devices. The kernel was also statically configured to contain device-specific information such as interrupt-vector locations and the locations of device-control registers on whatever bus the device was attached to.

This static approach was straightforward, but suffered from not being easily extensible. It required that a kernel be custom configured for each installation. The first approach taken to improving it was to still require that a kernel contain (i.e., be statically linked with) all necessary device drivers, but to allow the devices themselves to be found and automatically configured when the system booted. This was accomplished by having drivers include a *probe routine*, called at boot time. They would probe the relevant buses for devices and configure them, including identifying and recording interrupt-vector and device-control-register locations. This allowed one kernel image to be built that could be used for a number of similar but not identical installations.

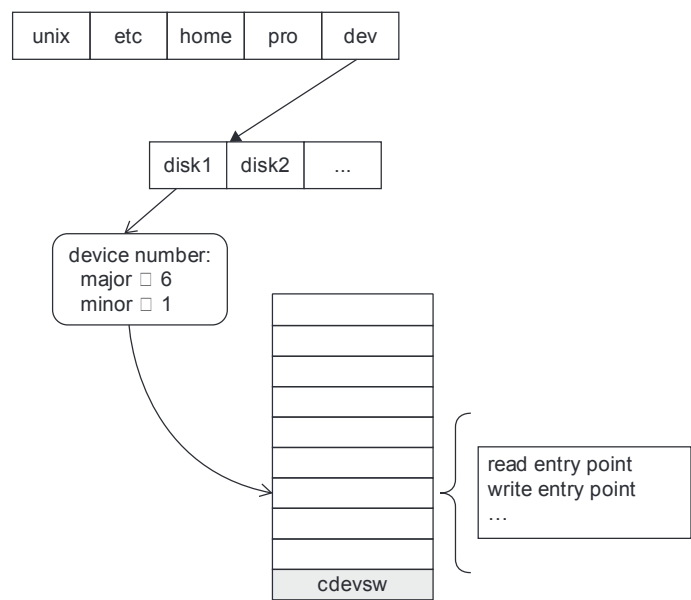


FIGURE 4.3 Devices in early Unix systems were named at the user level via special files containing the major and minor device numbers identifying the driver and the device. The major device number was an index into the *cdevsw* (character device switch), each of whose elements listed the entry points of a driver. The driver, in turn, was passed the minor device number and used it to determine the device.

² Unix drivers could support two interfaces: one for buffered I/O, called the *block* interface, and the other for nonbuffered I/O, called the *character* interface. There was also a *bdevsw* (block device switch), whose function was similar to the *cdevsw*. At the level of our discussion here, the two are identical.

There are zillions of devices available for today's systems. It is not practical to statically link all their drivers into an operating-system kernel, then check to see which devices are actually present. Instead, we must somehow discover devices without the benefit of having the relevant device driver in the kernel, then find the needed device drivers and dynamically link them into the kernel. This is a tall order, partly because identifying a device is a device-specific action.

The way around this conundrum is to allow meta-drivers that, for example, handle a particular kind of bus. Thus systems employing USB (universal serial bus) have a USB driver in addition to drivers for the various classes of devices that may be connected via USB. The USB protocol, interpreted by the USB driver, identifies the devices connected to it so that the appropriate device drivers can be selected. It is then up to system software (probably running at user level) to find the drivers and arrange for them to be loaded into the kernel.

A remaining issue is how to arrange for application programs to reference dynamically discovered devices. One approach, used until relatively recently on some Linux systems, was to list standard names for all possible devices in the `/dev` directory ((Kroah-Hartman 2003) reports that Red Hat 9 — a Linux distribution — has over eighteen thousand entries in its `/dev` directory). However, only those names associated with connected devices actually worked. Of course, if a device is added to a running system, its name would immediately be activated. An obvious improvement is to list in `/dev` only those names that are associated with connected devices. This was done in a later Linux distribution by having the kernel create entries in `/dev` as devices are discovered, getting their names from a database of names known as *devfs*. This is definitely a major advance, but it has the undesired property of imposing naming conventions that are not universally accepted. As of this writing, the current favorite Linux approach is known as *udev* (Kroah-Hartman 2003), in which a user-level application is notified by the kernel of new devices and assigns them names based on rules provided by an administrator.

A related problem is identifying which of a class of related devices to use. For example, you might have a laptop computer that has a touchpad. You then plug in a mouse. These are very different devices but have a similar use. How should the choice be presented to applications? Windows has the notion of *interface classes*, which allow devices to register themselves as members of one or more such classes. An application can enumerate all currently connected members of such a class and choose among them (or use them all).

4.1.2 LOW-LEVEL KERNEL

We start at the lowest levels of the kernel, showing how devices are handled. Though this bottom-up approach probably isn't how you would actually design an operating system, it's a reasonable order for implementing its components. Once lower-level modules are written and debugged, higher-level ones can be built on top of them.

4.1.2.1 Terminals

Terminals are long-obsolete devices (it is a good bet that most readers of this book have never seen one, let alone used one). We include them in our simple system both because of their simplicity and because much of the processing required of them is still present in current systems. After we discuss how they are handled, we talk about why this discussion is still relevant.

How should terminals be handled? This would seem pretty straightforward: characters to be output are simply sent to the output routine of the serial-line driver. To fetch characters that have been typed at the keyboard, a call is made to its input routine. However, this turns out to be an amazing oversimplification. Here are a few concerns that must be dealt with.

1. Characters are generated by the application faster than they can be sent to the terminal. If the application is continuously generating output at this high rate, then it clearly must be made to wait until the terminal can catch up with it. But if its output is bursty, then, rather

