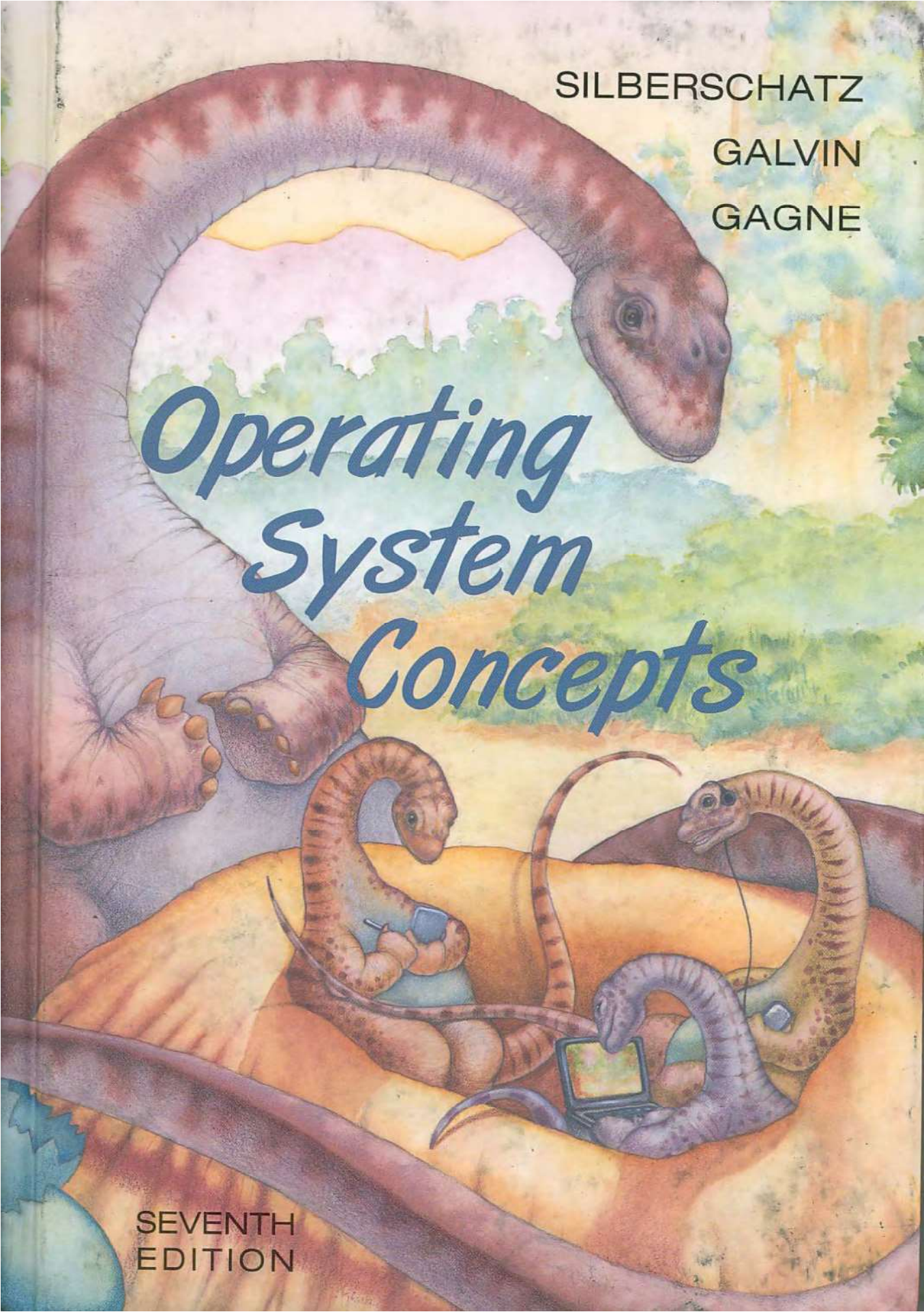




SILBERSCHATZ

GALVIN

GAGNE

Operating System Concepts

SEVENTH
EDITION

Operating System Concepts

Seventh Edition

ABRAHAM SILBERSCHATZ
Yale University

PETER BAER GALVIN
Corporate Technologies, Inc.

GREG GAGNE
Westminster College



JOHN WILEY & SONS, INC

EXECUTIVE EDITOR	Bill Zobrist
SENIOR PRODUCTION EDITOR	Ken Santor
COVER DESIGNER	Madelyn Lesure
COVER ILLUSTRATION	Susan St. Cyr
TEXT DESIGNER	Judy Allan

This book was set in Palatino by the author using LaTeX and printed and bound by Von Hoffmann, Inc. The cover was printed by Von Hoffmann, Inc.

This book is printed on acid free paper. ∞

Copyright © 2005 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, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008.

To order books or for customer service please, call 1(800)-CALL-WILEY (225-5945).

ISBN 0-471-69466-5

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Preface

Operating systems are an essential part of any computer system. Similarly, a course on operating systems is an essential part of any computer-science education. This field is undergoing rapid change, as computers are now prevalent in virtually every application, from games for children through the most sophisticated planning tools for governments and multinational firms. Yet the fundamental concepts remain fairly clear, and it is on these that we base this book.

We wrote this book as a text for an introductory course in operating systems at the junior or senior undergraduate level or at the first-year graduate level. We hope that practitioners will also find it useful. It provides a clear description of the *concepts* that underlie operating systems. As prerequisites, we assume that the reader is familiar with basic data structures, computer organization, and a high-level language, such as C. The hardware topics required for an understanding of operating systems are included in Chapter 1. For code examples, we use predominantly C, with some Java, but the reader can still understand the algorithms without a thorough knowledge of these languages.

Concepts are presented using intuitive descriptions. Important theoretical results are covered, but formal proofs are omitted. The bibliographical notes contain pointers to research papers in which results were first presented and proved, as well as references to material for further reading. In place of proofs, figures and examples are used to suggest why we should expect the result in question to be true.

The fundamental concepts and algorithms covered in the book are often based on those used in existing commercial operating systems. Our aim is to present these concepts and algorithms in a general setting that is not tied to one particular operating system. We present a large number of examples that pertain to the most popular and the most innovative operating systems, including Sun Microsystems' Solaris; Linux; Mach; Microsoft MS-DOS, Windows NT, Windows 2000, and Windows XP; DEC VMS and TOPS-20; IBM OS/2; and Apple Mac OS X.

In this text, when we refer to Windows XP as an example operating system, we are implying both Windows XP and Windows 2000. If a feature exists in Windows XP that is not available in Windows 2000, we will state this explicitly.

If a feature exists in Windows 2000 but not in Windows XP, then we will refer specifically to Windows 2000.

Organization of This Book

The organization of this text reflects our many years of teaching operating systems courses. Consideration was also given to the feedback provided by the reviewers of the text, as well as comments submitted by readers of earlier editions. In addition, the content of the text corresponds to the suggestions from *Computing Curricula 2001* for teaching operating systems, published by the Joint Task Force of the IEEE Computing Society and the Association for Computing Machinery (ACM).

On the supporting web page for this text, we provide several sample syllabi that suggest various approaches for using the text in both introductory and advanced operating systems courses. As a general rule, we encourage readers to progress sequentially through the chapters, as this strategy provides the most thorough study of operating systems. However, by using the sample syllabi, a reader can select a different ordering of chapters (or subsections of chapters).

Content of This Book

The text is organized in eight major parts:

- **Overview.** Chapters 1 and 2 explain what operating systems *are*, what they *do*, and how they are *designed* and *constructed*. They discuss what the common features of an operating system are, what an operating system does for the user, and what it does for the computer-system operator. The presentation is motivational and explanatory in nature. We have avoided a discussion of how things are done internally in these chapters. Therefore, they are suitable for individual readers or for students in lower-level classes who want to learn what an operating system is without getting into the details of the internal algorithms.
- **Process management.** Chapters 3 through 7 describe the process concept and concurrency as the heart of modern operating systems. A *process* is the unit of work in a system. Such a system consists of a collection of *concurrently* executing processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). These chapters cover methods for process scheduling, interprocess communication, process synchronization, and deadlock handling. Also included under this topic is a discussion of threads.
- **Memory management.** Chapters 8 and 9 deal with main memory management during the execution of a process. To improve both the utilization of the CPU and the speed of its response to its users, the computer must keep several processes in memory. There are many different memory-management schemes, reflecting various approaches to memory management, and the effectiveness of a particular algorithm depends on the situation.

- **Storage management.** Chapters 10 through 13 describe how the file system, mass storage, and I/O are handled in a modern computer system. The file system provides the mechanism for on-line storage of and access to both data and programs residing on the disks. These chapters describe the classic internal algorithms and structures of storage management. They provide a firm practical understanding of the algorithms used—the properties, advantages, and disadvantages. Since the I/O devices that attach to a computer vary widely, the operating system needs to provide a wide range of functionality to applications to allow them to control all aspects of the devices. We discuss system I/O in depth, including I/O system design, interfaces, and internal system structures and functions. In many ways, I/O devices are also the slowest major components of the computer. Because they are a performance bottleneck, performance issues are examined. Matters related to secondary and tertiary storage are explained as well.
- **Protection and security.** Chapters 14 and 15 discuss the processes in an operating system that must be protected from one another's activities. For the purposes of protection and security, we use mechanisms that ensure that only processes that have gained proper authorization from the operating system can operate on the files, memory, CPU, and other resources. Protection is a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means of specifying the controls to be imposed, as well as a means of enforcement. Security protects the information stored in the system (both data and code), as well as the physical resources of the computer system, from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency.
- **Distributed systems.** Chapters 16 through 18 deal with a collection of processors that do not share memory or a clock—a *distributed system*. By providing the user with access to the various resources that it maintains, a distributed system can improve computation speed and data availability and reliability. Such a system also provides the user with a distributed file system, which is a file-service system whose users, servers, and storage devices are dispersed among the sites of a distributed system. A distributed system must provide various mechanisms for process synchronization and communication and for dealing with the deadlock problem and a variety of failures that are not encountered in a centralized system.
- **Special-purpose systems.** Chapters 19 and 20 deal with systems used for specific purposes, including real-time systems and multimedia systems. These systems have specific requirements that differ from those of the general-purpose systems that are the focus of the remainder of the text. Real-time systems may require not only that computed results be “correct” but also that the results be produced within a specified deadline period. Multimedia systems require quality-of-service guarantees ensuring that the multimedia data are delivered to clients within a specific time frame.
- **Case studies.** Chapters 21 through 23 in the book, and Appendices A through C on the website, integrate the concepts described in this book by describing real operating systems. These systems include Linux, Windows

XP, FreeBSD, Mach, and Windows 2000. We chose Linux and FreeBSD because UNIX—at one time—was almost small enough to understand yet was not a “toy” operating system. Most of its internal algorithms were selected for *simplicity*, rather than for speed or sophistication. Both Linux and FreeBSD are readily available to computer-science departments, so many students have access to these systems. We chose Windows XP and Windows 2000 because they provide an opportunity for us to study a modern operating system with a design and implementation drastically different from those of UNIX. Chapter 23 briefly describes a few other influential operating systems.

Operating-System Environments

This book uses examples of many real-world operating systems to illustrate fundamental operating-system concepts. However, particular attention is paid to the Microsoft family of operating systems (including Windows NT, Windows 2000, and Windows XP) and various versions of UNIX (including Solaris, BSD, and Mac OS X). We also provide a significant amount of coverage of the Linux operating system reflecting the most recent version of the kernel—Version 2.6—at the time this book was written.

The text also provides several example programs written in C and Java. These programs are intended to run in the following programming environments:

- **Windows systems.** The primary programming environment for Windows systems is the Win32 API (application programming interface), which provides a comprehensive set of functions for managing processes, threads, memory, and peripheral devices. We provide several C programs illustrating the use of the Win32 API. Example programs were tested on systems running Windows 2000 and Windows XP.
- **POSIX.** POSIX (which stands for *Portable Operating System Interface*) represents a set of standards implemented primarily for UNIX-based operating systems. Although Windows XP and Windows 2000 systems can also run certain POSIX programs, our coverage of POSIX focuses primarily on UNIX and Linux systems. POSIX-compliant systems must implement the POSIX core standard (POSIX.1)—Linux, Solaris, and Mac OS X are examples of POSIX-compliant systems. POSIX also defines several extensions to the standards, including real-time extensions (POSIX.1.b) and an extension for a threads library (POSIX.1.c, better known as Pthreads). We provide several programming examples written in C illustrating the POSIX base API, as well as Pthreads and the extensions for real-time programming. These example programs were tested on Debian Linux 2.4 and 2.6 systems, Mac OS X, and Solaris 9 using the gcc 3.3 compiler.
- **Java.** Java is a widely used programming language with a rich API and built-in language support for thread creation and management. Java programs run on any operating system supporting a Java virtual machine (or JVM). We illustrate various operating system and networking concepts with several Java programs tested using the Java 1.4 JVM.

We have chosen these three programming environments because it is our opinion that they best represent the two most popular models of operating systems: Windows and UNIX/Linux, along with the widely used Java environment. Most programming examples are written in C, and we expect readers to be comfortable with this language; readers familiar with both the C and Java languages should easily understand most programs provided in this text.

In some instances—such as thread creation—we illustrate a specific concept using all three programming environments, allowing the reader to contrast the three different libraries as they address the same task. In other situations, we may use just one of the APIs to demonstrate a concept. For example, we illustrate shared memory using just the POSIX API; socket programming in TCP/IP is highlighted using the Java API.

The Seventh Edition

As we wrote this seventh edition of *Operating System Concepts*, we were guided by the many comments and suggestions we received from readers of our previous editions, as well as by our own observations about the rapidly changing fields of operating systems and networking. We have rewritten the material in most of the chapters by bringing older material up to date and removing material that was no longer of interest or relevance.

We have made substantive revisions and organizational changes in many of the chapters. Most importantly, we have completely reorganized the overview material in Chapters 1 and 2 and have added two new chapters on special-purpose systems (real-time embedded systems and multimedia systems). Because protection and security have become more prevalent in operating systems, we now cover these topics earlier in the text. Moreover, we have substantially updated and expanded the coverage of security.

Below, we provide a brief outline of the major changes to the various chapters:

- **Chapter 1, Introduction**, has been totally revised. In previous editions, the chapter gave a historical view of the development of operating systems. The new chapter provides a grand tour of the major operating-system components, along with basic coverage of computer-system organization.
- **Chapter 2, Operating-System Structures**, is a revised version of old Chapter 3, with many additions, including enhanced discussions of system calls and operating-system structure. It also provides significantly updated coverage of virtual machines.
- **Chapter 3, Processes**, is the old Chapter 4. It includes new coverage of how processes are represented in Linux and illustrates process creation using both the POSIX and Win32 APIs. Coverage of shared memory is enhanced with a program illustrating the shared-memory API available for POSIX systems.
- **Chapter 4, Threads**, is the old Chapter 5. The chapter presents an enhanced discussion of thread libraries, including the POSIX, Win32 API, and Java thread libraries. It also provides updated coverage of threading in Linux.

- **Chapter 5, CPU Scheduling**, is the old Chapter 6. The chapter offers a significantly updated discussion of scheduling issues for multiprocessor systems, including processor affinity and load-balancing algorithms. It also features a new section on thread scheduling, including Pthreads, and updated coverage of table-driven scheduling in Solaris. The section on Linux scheduling has been revised to cover the scheduler used in the 2.6 kernel.
- **Chapter 6, Process Synchronization**, is the old Chapter 7. We have removed the coverage of two-process solutions and now discuss only Peterson's solution, as the two-process algorithms are not guaranteed to work on modern processors. The chapter also includes new sections on synchronization in the Linux kernel and in the Pthreads API.
- **Chapter 7, Deadlocks**, is the old Chapter 8. New coverage includes a program example illustrating deadlock in a multithreaded Pthread program.
- **Chapter 8, Main Memory**, is the old Chapter 9. The chapter no longer covers overlays. In addition, the coverage of segmentation has seen significant modification, including an enhanced discussion of segmentation in Pentium systems and a discussion of how Linux is designed for such segmented systems.
- **Chapter 9, Virtual Memory**, is the old Chapter 10. The chapter features expanded coverage of motivating virtual memory as well as coverage of memory-mapped files, including a programming example illustrating shared memory (via memory-mapped files) using the Win32 API. The details of memory management hardware have been modernized. A new section on allocating memory within the kernel discusses the buddy algorithm and the slab allocator.
- **Chapter 10, File-System Interface**, is the old Chapter 11. It has been updated and an example of Windows XP ACLs has been added.
- **Chapter 11, File-System Implementation**, is the old Chapter 12. Additions include a full description of the WAFL file system and inclusion of Sun's ZFS file system.
- **Chapter 12, Mass-Storage Structure**, is the old Chapter 14. New is the coverage of modern storage arrays, including new RAID technology and features such as thin provisioning.
- **Chapter 13, I/O Systems**, is the old Chapter 13 updated with coverage of new material.
- **Chapter 14, Protection**, is the old Chapter 18 updated with coverage of the principle of least privilege.
- **Chapter 15, Security**, is the old Chapter 19. The chapter has undergone a major overhaul, with all sections updated. A full example of a buffer-overflow exploit is included, and coverage of threats, encryption, and security tools has been expanded.
- **Chapters 16 through 18** are the old Chapters 15 through 17, updated with coverage of new material.

- **Chapter 19, Real-Time Systems**, is a new chapter focusing on real-time and embedded computing systems, which have requirements different from those of many traditional systems. The chapter provides an overview of real-time computer systems and describes how operating systems must be constructed to meet the stringent timing deadlines of these systems.
- **Chapter 20, Multimedia Systems**, is a new chapter detailing developments in the relatively new area of multimedia systems. Multimedia data differ from conventional data in that multimedia data—such as frames of video—must be delivered (streamed) according to certain time restrictions. The chapter explores how these requirements affect the design of operating systems.
- **Chapter 21, The Linux System**, is the old Chapter 20, updated to reflect changes in the 2.6 kernel—the most recent kernel at the time this text was written.
- **Chapter 22, XP**, has been updated.
- **Chapter 22, Influential Operating Systems**, has been updated.

The old Chapter 21 (Windows 2000) has been turned into **Appendix C**. As in the previous edition, the appendices are provided online.

Programming Exercises and Projects

To emphasize the concepts presented in the text, we have added several programming exercises and projects that use the POSIX and Win32 APIs as well as Java. We have added over 15 new programming exercises that emphasize processes, threads, shared memory, process synchronization, and networking. In addition, we have added several programming projects which are more involved than standard programming exercises. These projects include adding a system call to the Linux kernel, creating a UNIX shell using the `fork()` system call, a multithreaded matrix application, and the producer-consumer problem using shared memory.

Teaching Supplements and Web Page

The web page for the book contains such material as a set of slides to accompany the book, model course syllabi, all C and Java source code, and up-to-date errata. The web page also contains the book's three case-study appendices and the Distributed Communication appendix. The URL is:

<http://www.os-book.com>

New to this edition is a print supplement called the Student Solutions Manual. Included are problems and exercises with solutions not found in the text that should help students master the concepts presented. You can purchase a print copy of this supplement at Wiley's website by going to <http://www.wiley.com/college/silberschatz> and choosing the Student Solutions Manual link.

To obtain restricted supplements, such as the solution guide to the exercises in the text, contact your local John Wiley & Sons sales representative. Note that these supplements are available only to faculty who use this text. You can find your representative at the “Find a Rep?” web page: <http://www.jswedcv.wiley.com/college/findarep>.

Mailing List

We have switched to the mailman system for communication among the users of *Operating System Concepts*. If you wish to use this facility, please visit the following URL and follow the instructions there to subscribe:

<http://mailman.cs.yale.edu/mailman/listinfo/os-book-list>

The mailman mailing-list system provides many benefits, such as an archive of postings, as well as several subscription options, including digest and Web only. To send messages to the list, send e-mail to:

os-book-list@cs.yale.edu

Depending on the message, we will either reply to you personally or forward the message to everyone on the mailing list. The list is moderated, so you will receive no inappropriate mail.

Students who are using this book as a text for class should not use the list to ask for answers to the exercises. They will not be provided.

Suggestions

We have attempted to clean up every error in this new edition, but—as happens with operating systems—a few obscure bugs may remain. We would appreciate hearing from you about any textual errors or omissions that you identify.

If you would like to suggest improvements or to contribute exercises, we would also be glad to hear from you. Please send correspondence to os-book@cs.yale.edu.

Acknowledgments

This book is derived from the previous editions, the first three of which were coauthored by James Peterson. Others who helped us with previous editions include Hamid Arabnia, Rida Bazzi, Randy Bentson, David Black, Joseph Boykin, Jeff Brumfield, Gael Buckley, Roy Campbell, P. C. Capon, John Carpenter, Gil Carrick, Thomas Casavant, Ajoy Kumar Datta, Joe Deck, Sudarshan K. Dhall, Thomas Doeppner, Caleb Drake, M. Racsit Eskicioğlu, Hans Flack, Robert Fowler, G. Scott Graham, Richard Guy, Max Hailperin, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Bruce Hillyer, Mark Holliday, Ahmed Kamel, Richard Kieburtz, Carol Kroll, Morty Kwestel, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller,

Michael Molloy, Yoichi Muraoka, Jim M. Ng, Banu Özden, Ed Posnak, Boris Putanec, Charles Qualline, John Quarterman, Mike Reiter, Gustavo Rodriguez-Rivera, Carolyn J. C. Schauble, Thomas P. Skinner, Yannis Smaragdakis, Jesse St. Laurent, John Stankovic, Adam Stauffer, Steven Stepanek, Hal Stern, Louis Stevens, Pete Thomas, David Umbaugh, Steve Vinoski, Tommy Wagner, Larry L. Wear, John Werth, James M. Westall, J. S. Weston, and Yang Xiang

Parts of Chapter 12 were derived from a paper by Hillyer and Silberschatz [1996]. Parts of Chapter 17 were derived from a paper by Levy and Silberschatz [1990]. Chapter 21 was derived from an unpublished manuscript by Stephen Tweedie. Chapter 22 was derived from an unpublished manuscript by Dave Probert, Cliff Martin, and Avi Silberschatz. Appendix C was derived from an unpublished manuscript by Cliff Martin. Cliff Martin also helped with updating the UNIX appendix to cover FreeBSD. Mike Shapiro, Bryan Cantrill, and Jim Mauro answered several Solaris-related questions. Josh Dees and Rob Reynolds contributed coverage of Microsoft's .NET. The project for designing and enhancing the UNIX shell interface was contributed by John Trono of St. Michael's College in Winooski, Vermont.

This edition has many new exercises and accompanying solutions, which were supplied by Arvind Krishnamurthy.

We thank the following people who reviewed this version of the book: Bart Childs, Don Heller, Dean Hougen, Michael Huang, Morty Kewstel, Euripides Montagne, and John Sterling.

Our Acquisitions Editors, Bill Zobrist and Paul Crockett, provided expert guidance as we prepared this edition. They were assisted by Simon Durkin, who managed many details of this project smoothly. The Senior Production Editor was Ken Santor. The cover illustrator was Susan Cyr, and the cover designer was Madelyn Lesure. Beverly Peavler copy-edited the manuscript. The freelance proofreader was Katrina Avery; the freelance indexer was Rosemary Simpson. Marilyn Turnamian helped generate figures and presentation slides.

Finally, we would like to add some personal notes. Avi is starting a new chapter in his life, returning to academia and partnering with Valerie. This combination has given him the peace of mind to focus on the writing of this text. Pete would like to thank his family, friends, and coworkers for their support and understanding during the project. Greg would like to acknowledge the continued interest and support from his family. However, he would like to single out his friend Peter Ormsby who—no matter how busy his life seems to be—always first asks, “How’s the writing coming along?”

Abraham Silberschatz, New Haven, CT, 2004

Peter Baer Galvin, Burlington, MA, 2004

Greg Gagne, Salt Lake City, UT, 2004

Contents

PART ONE ■ OVERVIEW

Chapter 1 Introduction

- | | | | |
|----------------------------------|----|------------------------------|----|
| 1.1 What Operating Systems Do | 3 | 1.9 Protection and Security | 26 |
| 1.2 Computer-System Organization | 6 | 1.10 Distributed Systems | 28 |
| 1.3 Computer-System Architecture | 12 | 1.11 Special-Purpose Systems | 29 |
| 1.4 Operating-System Structure | 15 | 1.12 Computing Environments | 31 |
| 1.5 Operating-System Operations | 17 | 1.13 Summary | 34 |
| 1.6 Process Management | 20 | Exercises | 36 |
| 1.7 Memory Management | 21 | Bibliographical Notes | 38 |
| 1.8 Storage Management | 22 | | |

Chapter 2 Operating-System Structures

- | | | | |
|--|----|---------------------------------|----|
| 2.1 Operating-System Services | 39 | 2.7 Operating-System Structure | 58 |
| 2.2 User Operating-System Interface | 41 | 2.8 Virtual Machines | 64 |
| 2.3 System Calls | 43 | 2.9 Operating-System Generation | 70 |
| 2.4 Types of System Calls | 47 | 2.10 System Boot | 71 |
| 2.5 System Programs | 55 | 2.11 Summary | 72 |
| 2.6 Operating-System Design and Implementation | 56 | Exercises | 73 |
| | | Bibliographical Notes | 78 |

PART TWO ■ PROCESS MANAGEMENT

Chapter 3 Processes

- | | | | |
|--------------------------------|-----|--|-----|
| 3.1 Process Concept | 81 | 3.6 Communication in Client-Server Systems | 108 |
| 3.2 Process Scheduling | 85 | 3.7 Summary | 115 |
| 3.3 Operations on Processes | 90 | Exercises | 116 |
| 3.4 Interprocess Communication | 96 | Bibliographical Notes | 125 |
| 3.5 Examples of IPC Systems | 102 | | |

Chapter 4 Threads

- 4.1 Overview 127
- 4.2 Multithreading Models 129
- 4.3 Thread Libraries 131
- 4.4 Threading Issues 138
- 4.5 Operating-System Examples 143
- 4.6 Summary 146
 - Exercises 146
 - Bibliographical Notes 151

Chapter 5 CPU Scheduling

- 5.1 Basic Concepts 153
- 5.2 Scheduling Criteria 157
- 5.3 Scheduling Algorithms 158
- 5.4 Multiple-Processor Scheduling 169
- 5.5 Thread Scheduling 172
- 5.6 Operating System Examples 173
- 5.7 Algorithm Evaluation 181
- 5.8 Summary 185
 - Exercises 186
 - Bibliographical Notes 189

Chapter 6 Process Synchronization

- 6.1 Background 191
- 6.2 The Critical-Section Problem 193
- 6.3 Peterson's Solution 195
- 6.4 Synchronization Hardware 197
- 6.5 Semaphores 200
- 6.6 Classic Problems of Synchronization 204
- 6.7 Monitors 209
- 6.8 Synchronization Examples 217
- 6.9 Atomic Transactions 222
- 6.10 Summary 230
 - Exercises 231
 - Bibliographical Notes 242

Chapter 7 Deadlocks

- 7.1 System Model 245
- 7.2 Deadlock Characterization 247
- 7.3 Methods for Handling Deadlocks 252
- 7.4 Deadlock Prevention 253
- 7.5 Deadlock Avoidance 256
- 7.6 Deadlock Detection 262
- 7.7 Recovery From Deadlock 266
- 7.8 Summary 267
 - Exercises 268
 - Bibliographical Notes 271

PART THREE ■ MEMORY MANAGEMENT**Chapter 8 Main Memory**

- 8.1 Background 275
- 8.2 Swapping 282
- 8.3 Contiguous Memory Allocation 284
- 8.4 Paging 288
- 8.5 Structure of the Page Table 297
- 8.6 Segmentation 302
- 8.7 Example: The Intel Pentium 305
- 8.8 Summary 309
 - Exercises 310
 - Bibliographical Notes 312

Chapter 9 Virtual Memory

- 9.1 Background 315
- 9.2 Demand Paging 319
- 9.3 Copy-on-Write 325
- 9.4 Page Replacement 327
- 9.5 Allocation of Frames 340
- 9.6 Thrashing 343
- 9.7 Memory-Mapped Files 348
- 9.8 Allocating Kernel Memory 353
- 9.9 Other Considerations 357
- 9.10 Operating-System Examples 363
- 9.11 Summary 365
 - Exercises 366
 - Bibliographical Notes 370

PART FOUR ■ STORAGE MANAGEMENT

Chapter 10 File-System Interface

- 10.1 File Concept 373
- 10.2 Access Methods 382
- 10.3 Directory Structure 385
- 10.4 File-System Mounting 395
- 10.5 File Sharing 397
- 10.6 Protection 402
- 10.7 Summary 407
 - Exercises 408
 - Bibliographical Notes 409

Chapter 11 File-System Implementation

- 11.1 File-System Structure 411
- 11.2 File-System Implementation 413
- 11.3 Directory Implementation 419
- 11.4 Allocation Methods 421
- 11.5 Free-Space Management 429
- 11.6 Efficiency and Performance 431
- 11.7 Recovery 435
- 11.8 Log-Structured File Systems 437
- 11.9 NFS 438
- 11.10 Example: The WAFL File System 444
- 11.11 Summary 446
 - Exercises 447
 - Bibliographical Notes 449

Chapter 12 Mass-Storage Structure

- 12.1 Overview of Mass-Storage Structure 451
- 12.2 Disk Structure 454
- 12.3 Disk Attachment 455
- 12.4 Disk Scheduling 456
- 12.5 Disk Management 462
- 12.6 Swap-Space Management 466
- 12.7 RAID Structure 468
- 12.8 Stable-Storage Implementation 477
- 12.9 Tertiary-Storage Structure 478
- 12.10 Summary 488
 - Exercises 489
 - Bibliographical Notes 493

Chapter 13 I/O Systems

- 13.1 Overview 495
- 13.2 I/O Hardware 496
- 13.3 Application I/O Interface 505
- 13.4 Kernel I/O Subsystem 511
- 13.5 Transforming I/O Requests to Hardware Operations 518
- 13.6 STREAMS 520
- 13.7 Performance 522
- 13.8 Summary 525
 - Exercises 526
 - Bibliographical Notes 527

PART FIVE ■ PROTECTION AND SECURITY

Chapter 14 Protection

- | | | | |
|--------------------------------------|-----|----------------------------------|-----|
| 14.1 Goals of Protection | 531 | 14.7 Revocation of Access Rights | 546 |
| 14.2 Principles of Protection | 532 | 14.8 Capability-Based Systems | 547 |
| 14.3 Domain of Protection | 533 | 14.9 Language-Based Protection | 550 |
| 14.4 Access Matrix | 538 | 14.10 Summary | 555 |
| 14.5 Implementation of Access Matrix | 542 | Exercises | 556 |
| 14.6 Access Control | 545 | Bibliographical Notes | 557 |

Chapter 15 Security

- | | | | |
|--|-----|--|-----|
| 15.1 The Security Problem | 559 | 15.8 Computer-Security Classifications | 600 |
| 15.2 Program Threats | 563 | 15.9 An Example: Windows XP | 602 |
| 15.3 System and Network Threats | 571 | 15.10 Summary | 604 |
| 15.4 Cryptography as a Security Tool | 576 | Exercises | 604 |
| 15.5 User Authentication | 587 | Bibliographical Notes | 606 |
| 15.6 Implementing Security Defenses | 592 | | |
| 15.7 Firewalling to Protect Systems and Networks | 599 | | |

PART SIX ■ DISTRIBUTED SYSTEMS

Chapter 16 Distributed System Structures

- | | | | |
|---|-----|-----------------------------|-----|
| 16.1 Motivation | 611 | 16.7 Robustness | 631 |
| 16.2 Types of Distributed Operating Systems | 613 | 16.8 Design Issues | 633 |
| 16.3 Network Structure | 617 | 16.9 An Example: Networking | 636 |
| 16.4 Network Topology | 620 | 16.10 Summary | 637 |
| 16.5 Communication Structure | 622 | Exercises | 638 |
| 16.6 Communication Protocols | 628 | Bibliographical Notes | 640 |

Chapter 17 Distributed File Systems

- | | | | |
|--|-----|-----------------------|-----|
| 17.1 Background | 641 | 17.6 An Example: AFS | 654 |
| 17.2 Naming and Transparency | 643 | 17.7 Summary | 659 |
| 17.3 Remote File Access | 646 | Exercises | 660 |
| 17.4 Stateful Versus Stateless Service | 651 | Bibliographical Notes | 661 |
| 17.5 File Replication | 652 | | |

Chapter 18 Distributed Coordination

- | | |
|------------------------------|------------------------------|
| 18.1 Event Ordering 663 | 18.6 Election Algorithms 683 |
| 18.2 Mutual Exclusion 666 | 18.7 Reaching Agreement 686 |
| 18.3 Atomicity 669 | 18.8 Summary 688 |
| 18.4 Concurrency Control 672 | Exercises 689 |
| 18.5 Deadlock Handling 676 | Bibliographical Notes 690 |

PART SEVEN ■ SPECIAL-PURPOSE SYSTEMS**Chapter 19 Real-Time Systems**

- | | |
|---|-----------------------------------|
| 19.1 Overview 695 | 19.5 Real-Time CPU Scheduling 704 |
| 19.2 System Characteristics 696 | 19.6 VxWorks 5.x 710 |
| 19.3 Features of Real-Time Kernels 698 | 19.7 Summary 712 |
| 19.4 Implementing Real-Time Operating Systems 700 | Exercises 713 |
| | Bibliographical Notes 713 |

Chapter 20 Multimedia Systems

- | | |
|---|--------------------------------|
| 20.1 What Is Multimedia? 715 | 20.6 Network Management 725 |
| 20.2 Compression 718 | 20.7 An Example: CineBlitz 728 |
| 20.3 Requirements of Multimedia Kernels 720 | 20.8 Summary 730 |
| 20.4 CPU Scheduling 722 | Exercises 731 |
| 20.5 Disk Scheduling 723 | Bibliographical Notes 733 |

PART EIGHT ■ CASE STUDIES**Chapter 21 The Linux System**

- | | |
|-----------------------------|-------------------------------------|
| 21.1 Linux History 737 | 21.8 Input and Output 770 |
| 21.2 Design Principles 742 | 21.9 Interprocess Communication 773 |
| 21.3 Kernel Modules 745 | 21.10 Network Structure 774 |
| 21.4 Process Management 748 | 21.11 Security 777 |
| 21.5 Scheduling 751 | 21.12 Summary 779 |
| 21.6 Memory Management 756 | Exercises 780 |
| 21.7 File Systems 764 | Bibliographical Notes 781 |

Chapter 22 Windows XP

- | | |
|-----------------------------------|-------------------------------|
| 22.1 History 783 | 22.6 Networking 822 |
| 22.2 Design Principles 785 | 22.7 Programmer Interface 829 |
| 22.3 System Components 787 | 22.8 Summary 836 |
| 22.4 Environmental Subsystems 811 | Exercises 836 |
| 22.5 File System 814 | Bibliographical Notes 837 |

Chapter 23 Influential Operating Systems

- | | |
|------------------------|-------------------------|
| 23.1 Early Systems 839 | 23.7 MULTICS 849 |
| 23.2 Atlas 845 | 23.8 IBM OS/360 850 |
| 23.3 XDS-940 846 | 23.9 Mach 851 |
| 23.4 THE 847 | 23.10 Other Systems 853 |
| 23.5 RC 4000 848 | Exercises 853 |
| 23.6 CTSS 849 | |

Appendix A UNIX BSD (contents online)

- | | |
|-------------------------------|-------------------------------------|
| A.1 UNIX History A855 | A.7 File System A878 |
| A.2 Design Principles A860 | A.8 I/O System A886 |
| A.3 Programmer Interface A862 | A.9 Interprocess Communication A889 |
| A.4 User Interface A869 | A.10 Summary A894 |
| A.5 Process Management A872 | Exercises A895 |
| A.6 Memory Management A876 | Bibliographical Notes A896 |

Appendix B The Mach System (contents online)

- | | |
|-------------------------------------|-------------------------------|
| B.1 History of the Mach System A897 | B.7 Programmer Interface A919 |
| B.2 Design Principles A899 | B.8 Summary A920 |
| B.3 System Components A900 | Exercises A921 |
| B.4 Process Management A903 | Bibliographical Notes A922 |
| B.5 Interprocess Communication A909 | Credits A923 |
| B.6 Memory Management A914 | |

Appendix C Windows 2000 (contents online)

- | | |
|-----------------------------------|-------------------------------|
| C.1 History A925 | C.6 Networking A952 |
| C.2 Design Principles A926 | C.7 Programmer Interface A957 |
| C.3 System Components A927 | C.8 Summary A964 |
| C.4 Environmental Subsystems A943 | Exercises A964 |
| C.5 File System A945 | Bibliographical Notes A965 |

Bibliography 855

Credits 885

Index 887

Process Synchronization



A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of lightweight processes or threads, which we discussed in Chapter 4. Concurrent access to shared data may result in data inconsistency. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

CHAPTER OBJECTIVES

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.
- To present both software and hardware solutions of the critical-section problem.
- To introduce the concept of atomic transaction and describe mechanisms to ensure atomicity.

6.1 Background

In Chapter 3, we developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. We illustrated this model with the producer-consumer problem, which is representative of operating systems. Specifically, in Section 3.4.1, we described how a bounded buffer could be used to enable processes to share memory.

Let us return to our consideration of the bounded buffer. As we pointed out, our solution allows at most `BUFFER_SIZE - 1` items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable `counter`, initialized to 0. `counter` is incremented every time we add a new item to the buffer and is decremented

every time we remove one item from the buffer. The code for the producer process can be modified as follows:

```
while (true)
{
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

The code for the consumer process can be modified as follows:

```
while (true)
{
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in nextConsumed */
}
```

Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable `counter` is currently 5 and that the producer and consumer processes execute the statements “`counter++`” and “`counter--`” concurrently. Following the execution of these two statements, the value of the variable `counter` may be 4, 5, or 6! The only correct result, though, is `counter == 5`, which is generated correctly if the producer and consumer execute separately.

We can show that the value of `counter` may be incorrect as follows. Note that the statement “`counter++`” may be implemented in machine language (on a typical machine) as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

where `register1` is a local CPU register. Similarly, the statement “`counter--`” is implemented as follows:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

where again `register2` is a local CPU register. Even though `register1` and `register2` may be the same physical register (an accumulator, say), remember

that the contents of this register will be saved and restored by the interrupt handler (Section 1.2.3).

The concurrent execution of “counter++” and “counter--” is equivalent to a sequential execution where the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is

T_0 :	producer	execute	$register_1 = counter$	$\{register_1 = 5\}$
T_1 :	producer	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2 :	consumer	execute	$register_2 = counter$	$\{register_2 = 5\}$
T_3 :	consumer	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4 :	producer	execute	$counter = register_1$	$\{counter = 6\}$
T_5 :	consumer	execute	$counter = register_2$	$\{counter = 4\}$

Notice that we have arrived at the incorrect state “counter == 4”, indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T_4 and T_5 , we would arrive at the incorrect state “counter == 6”.

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Clearly, we want the resulting changes not to interfere with one another. Because of the importance of this issue, a major portion of this chapter is concerned with **process synchronization and coordination**.

6.2 The Critical-Section Problem

Consider a system consisting of n processes $\{P_0, P_1, \dots, P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process P_i is shown in Figure 6.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

```

do {
    entry section
    critical section
    exit section
    remainder section
} while (TRUE);

```

Figure 6.1 General structure of a typical process P_i .

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion.** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress.** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting.** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the **relative speed** of the n processes.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (*kernel code*) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

Two general approaches are used to handle critical sections in operating systems: (1) **preemptive kernels** and (2) **nonpreemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as

only one process is active in the kernel at a time. We cannot say the same about nonpreemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

Why, then, would anyone favor a preemptive kernel over a nonpreemptive one? A preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel. Furthermore, a preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. Of course, this effect can be minimized by designing kernel code that does not behave in this way.

Windows XP and Windows 2000 are nonpreemptive kernels, as is the traditional UNIX kernel. Prior to Linux 2.6, the Linux kernel was nonpreemptive as well. However, with the release of the 2.6 kernel, Linux changed to the preemptive model. Several commercial versions of UNIX are preemptive, including Solaris and IRIX.

6.3 Peterson's Solution

Next, we illustrate a classic software-based solution to the critical-section problem known as **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_0 and P_1 . For convenience, when presenting P_i , we use P_j to denote the other process; that is, j equals $1 - i$.

Peterson's solution requires two data items to be shared between the two processes:

```
int turn;
boolean flag[2];
```

The variable `turn` indicates whose turn it is to enter its critical section. That is, if `turn == i`, then process P_i is allowed to execute in its critical section. The `flag` array is used to indicate if a process *is ready* to enter its critical section. For example, if `flag[i]` is `true`, this value indicates that P_i is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure 6.2.

To enter the critical section, process P_i first sets `flag[i]` to be `true` and then sets `turn` to the value j , thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, `turn` will be set to both i and j at roughly the same time. Only

```

do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

    critical section

    flag[i] = FALSE;

    remainder section

} while (TRUE);

```

Figure 6.2 The structure of process P_i in Peterson's solution.

one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of `turn` decides which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property 1, we note that each P_i enters its critical section only if either `flag[j] == false` or `turn == i`. Also note that, if both processes can be executing in their critical sections at the same time, then `flag[0] == flag[1] == true`. These two observations imply that P_0 and P_1 could not have successfully executed their `while` statements at about the same time, since the value of `turn` can be either 0 or 1 but cannot be both. Hence, one of the processes—say P_j —must have successfully executed the `while` statement, whereas P_i had to execute at least one additional statement ("`turn == j`"). However, since, at that time, `flag[j] == true`, and `turn == j`, and this condition will persist as long as P_j is in its critical section, the result follows: Mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the `while` loop with the condition `flag[j] == true` and `turn == j`; this loop is the only one possible. If P_j is not ready to enter the critical section, then `flag[j] == false`, and P_i can enter its critical section. If P_j has set `flag[j]` to `true` and is also executing in its `while` statement, then either `turn == i` or `turn == j`. If `turn == i`, then P_i will enter the critical section. If `turn == j`, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset `flag[j]` to `false`, allowing P_i to enter its critical section. If P_j resets `flag[j]` to `true`, it must also set `turn` to `i`. Thus, since P_i does not change the value of the variable `turn` while executing the `while` statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

```

do {
    acquire lock

    critical section

    release lock

    remainder section
} while (TRUE);

```

Figure 6.3 Solution to the critical-section problem using locks.

6.4 Synchronization Hardware

We have just described one software-based solution to the critical-section problem. In general, we can state that any solution to the critical-section problem requires a simple tool—a **lock**. Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section. This is illustrated in Figure 6.3.

In the following discussions, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software-based APIs available to application programmers. All these solutions are based on the premise of locking; however, as we shall see, the design of such locks can be quite sophisticated.

Hardware features can make any programming task easier and improve system efficiency. In this section, we present some simple hardware instructions that are available on many systems and show how they can be used effectively in solving the critical-section problem.

The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is the approach taken by nonpreemptive kernels.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the

```

boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}

```

Figure 6.4 The definition of the TestAndSet() instruction.

```

do {
    while (TestAndSetLock(&lock))
        ; // do nothing

    // critical section

    lock = FALSE;

    // remainder section
}while (TRUE);

```

Figure 6.5 Mutual-exclusion implementation with TestAndSet().

message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also, consider the effect on a system's clock, if the clock is kept updated by interrupts.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically**—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions.

The TestAndSet() instruction can be defined as shown in Figure 6.4. The important characteristic is that this instruction is executed atomically. Thus, if two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet() instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process P_i is shown in Figure 6.5.

The Swap() instruction, in contrast to the TestAndSet() instruction, operates on the contents of two words; it is defined as shown in Figure 6.6. Like the TestAndSet() instruction, it is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows. A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key. The structure of process P_i is shown in Figure 6.7.

Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement. In Figure 6.8, we present

```

void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}

```

Figure 6.6 The definition of the Swap() instruction.

```

do {
    key = TRUE;
    while (key == TRUE)
        Swap(&lock, &key);

    // critical section

    lock = FALSE;

    // remainder section
}while (TRUE);

```

Figure 6.7 Mutual-exclusion implementation with the Swap() instruction.

another algorithm using the TestAndSet() instruction that satisfies all the critical-section requirements. The common data structures are

```

boolean waiting[n];
boolean lock;

```

These data structures are initialized to false. To prove that the mutual-exclusion requirement is met, we note that process P_i can enter its critical section only if either `waiting[i] == false` or `key == false`. The value of `key` can become false only if the TestAndSet() is executed. The first process to execute the TestAndSet() will find `key == false`; all others must

```

do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // critical section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

    // remainder section
}while (TRUE);

```

Figure 6.8 Bounded-waiting mutual exclusion with TestAndSet().

wait. The variable `waiting[i]` can become false only if another process leaves its critical section; only one `waiting[i]` is set to false, maintaining the mutual-exclusion requirement.

To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets `lock` to false or sets `waiting[j]` to false. Both allow a process that is waiting to enter its critical section to proceed.

To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array `waiting` in the cyclic ordering $(i + 1, i + 2, \dots, n - 1, 0, \dots, i - 1)$. It designates the first process in this ordering that is in the entry section (`waiting[j] == true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

Unfortunately for hardware designers, implementing atomic `TestAndSet()` instructions on multiprocessors is not a trivial task. Such implementations are discussed in books on computer architecture.

6.5 Semaphores

The various hardware-based solutions to the critical-section problem (using the `TestAndSet()` and `Swap()` instructions) presented in Section 6.4 are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a **semaphore**.

A semaphore `S` is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`. The `wait()` operation was originally termed *P* (from the Dutch *proberen*, “to test”); `signal()` was originally called *V* (from *verhogen*, “to increment”). The definition of `wait()` is as follows:

```
wait(S) {
    while S <= 0
        ; // no-op
    S--;
}
```

The definition of `signal()` is as follows:

```
signal(S) {
    S++;
}
```

All the modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of `wait(S)`, the testing of the integer value of `S` ($S \leq 0$), and its possible modification (`S--`), must also be executed without interruption. We shall see how these operations can be implemented in Section 6.5.2; first, let us see how semaphores can be used.

6.5.1 Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide *mutual exclusion*.

We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a semaphore, *mutex*, initialized to 1. Each process P_i is organized as shown in Figure 6.9.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a `wait()` operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a `signal()` operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2 . Suppose we require that S_2 be executed only after S_1 has completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore *synch*, initialized to 0, and by inserting the statements

```
S1;  
signal(synch);
```

in process P_1 , and the statements

```
wait(synch);  
S2;
```

in process P_2 . Because *synch* is initialized to 0, P_2 will execute S_2 only after P_1 has invoked `signal(synch)`, which is after statement S_1 has been executed.

```
do {  
    waiting(mutex);  
  
    // critical section  
  
    signal(mutex);  
  
    // remainder section  
}while (TRUE);
```

Figure 6.9 Mutual-exclusion implementation with semaphores.

6.5.2 Implementation

The main disadvantage of the semaphore definition given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** because the process “spins” while waiting for the lock. (Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful; they are often employed on multiprocessor systems where one thread can “spin” on one processor while another thread performs its critical section on another processor.)

To overcome the need for busy waiting, we can modify the definition of the `wait()` and `signal()` semaphore operations. When a process executes the `wait()` operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can *block* itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore `S`, should be restarted when some other process executes a `signal()` operation. The process is restarted by a `wakeup()` operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as a “C” struct:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes `list`. When a process must wait on a semaphore, it is added to the list of processes. A `signal()` operation removes one process from the list of waiting processes and awakens that process.

The `wait()` semaphore operation can now be defined as

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

The `signal()` semaphore operation can now be defined as

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

The `block()` operation suspends the process that invokes it. The `wakeup(P)` operation resumes the execution of a blocked process *P*. These two operations are provided by the operating system as basic system calls.

Note that, although under the classical definition of semaphores with busy waiting the semaphore value is never negative, this implementation may have negative semaphore values. If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the `wait()` operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list in a way that ensures bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use *any* queueing strategy. Correct usage of semaphores does not depend on a particular queueing strategy for the semaphore lists.

The critical aspect of semaphores is that they be executed atomically. We must guarantee that no two processes can execute `wait()` and `signal()` operations on the same semaphore at the same time. This is a critical-section problem; and in a single-processor environment (that is, where only one CPU exists), we can solve it by simply inhibiting interrupts during the time the `wait()` and `signal()` operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

In a multiprocessor environment, interrupts must be disabled on every processor; otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitrary way. Disabling interrupts on every processor can be a difficult task and furthermore can seriously diminish performance. Therefore, SMP systems must provide alternative locking techniques—such as spinlocks—to ensure that `wait()` and `signal()` are performed atomically.

It is important to admit that we have not completely eliminated busy waiting with this definition of the `wait()` and `signal()` operations. Rather, we have removed busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the `wait()` and `signal()` operations, and these sections are short (if properly coded, they should be no more than about ten instructions).

Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or even hours) or may almost always be occupied. In such cases, busy waiting is extremely inefficient.

6.5.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a `signal()` operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, we consider a system consisting of two processes, P_0 and P_1 , each accessing two semaphores, S and Q , set to the value 1:

P_0	P_1
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

Suppose that P_0 executes `wait(S)` and then P_1 executes `wait(Q)`. When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`. Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`. Since these `signal()` operations cannot be executed, P_0 and P_1 are deadlocked.

We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are *resource acquisition and release*. However, other types of events may result in deadlocks, as we shall show in Chapter 7. In that chapter, we shall describe various mechanisms for dealing with the deadlock problem.

Another problem related to deadlocks is **indefinite blocking**, or **starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

6.6 Classic Problems of Synchronization

In this section, we present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization.

```

do {
    . . .
    // produce an item in nextp
    . . .
    wait(empty);
    wait(mutex);
    . . .
    // add nextp to buffer
    . . .
    signal(mutex);
    signal(full);
}while (TRUE);

```

Figure 6.10 The structure of the producer process.

6.6.1 The Bounded-Buffer Problem

The *bounded-buffer problem* was introduced in Section 6.1; it is commonly used to illustrate the power of synchronization primitives. We present here a general structure of this scheme without committing ourselves to any particular implementation; we provide a related programming project in the exercises at the end of the chapter.

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n ; the semaphore full is initialized to the value 0.

The code for the producer process is shown in Figure 6.10; the code for the consumer process is shown in Figure 6.11. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

```

do {
    wait(full);
    wait(mutex);
    . . .
    // remove an item from buffer to nextc
    . . .
    signal(mutex);
    signal(empty);
    . . .
    // consume the item in nextc
    . . .
}while (TRUE);

```

Figure 6.11 The structure of the consumer process.

6.6.2 The Readers–Writers Problem

A database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers**. Obviously, if two readers access the shared data simultaneously, no adverse affects will result. However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database. This synchronization problem is referred to as the *readers–writers problem*. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers–writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers–writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we present a solution to the first readers–writers problem. Refer to the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers–writers problem.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore mutex, wrt;
int readcount;
```

The semaphores `mutex` and `wrt` are initialized to 1; `readcount` is initialized to 0. The semaphore `wrt` is common to both reader and writer processes. The `mutex` semaphore is used to ensure mutual exclusion when the variable `readcount` is updated. The `readcount` variable keeps track of how many processes are currently reading the object. The semaphore `wrt` functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last

```
do {
    wait(wrt);
    . . .
    // writing is performed
    . . .
    signal(wrt);
}while (TRUE);
```

Figure 6.12 The structure of a writer process.

```

do {
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(wrt);
    signal(mutex);
    . . .
    // reading is performed
    . . .
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
}while (TRUE);

```

Figure 6.13 The structure of a reader process.

reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 6.12; the code for a reader process is shown in Figure 6.13. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on `wrt`, and $n - 1$ readers are queued on `mutex`. Also observe that, when a writer executes `signal(wrt)`, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers–writers problem and its solutions has been generalized to provide **reader–writer** locks on some systems. Acquiring a reader–writer lock requires specifying the mode of the lock: either *read* or *write* access. When a process only wishes to read shared data, it requests the reader–writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode; only one process may acquire the lock for writing as exclusive access is required for writers.

Reader–writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which threads only write shared data.
- In applications that have more readers than writers. This is because reader–writer locks generally require more overhead to establish than semaphores or mutual exclusion locks, and the overhead for setting up a reader–writer lock is compensated by the increased concurrency of allowing multiple readers.

6.6.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid

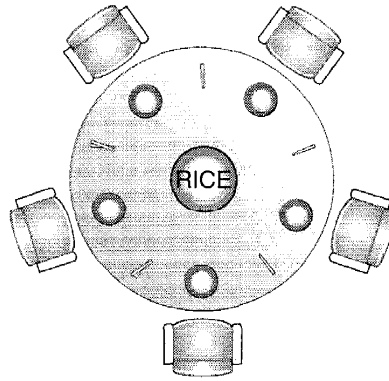


Figure 6.14 The situation of the dining philosophers.

with five single chopsticks (Figure 6.14). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The *dining-philosophers problem* is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore; she releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of `chopstick` are initialized to 1. The structure of philosopher i is shown in Figure 6.15.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of `chopstick` will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next. In Section 6.7, we present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.

```

do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    . . .
    // eat
    . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    . . .
    // think
    . . .
}while (TRUE);

```

Figure 6.15 The structure of philosopher *i*.

- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Finally, any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

6.7 Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.

We have seen an example of such errors in the use of counters in our solution to the producer–consumer problem (Section 6.1). In that example, the timing problem happened only rarely, and even then the counter value appeared to be reasonable—off by only 1. Nevertheless, the solution is obviously not an acceptable one. It is for this reason that semaphores were introduced in the first place.

Unfortunately, such timing errors can still occur when semaphores are used. To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a semaphore variable *mutex*, which is initialized to 1. Each process must execute *wait(mutex)* before entering the critical section and *signal(mutex)* afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. Let us examine the various difficulties that may result. Note that these difficulties will arise even if a *single* process is not well behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

- Suppose that a process interchanges the order in which the `wait()` and `signal()` operations on the semaphore `mutex` are executed, resulting in the following execution:

```

signal(mutex);
...
critical section
...
wait(mutex);

```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces `signal(mutex)` with `wait(mutex)`. That is, it executes

```

wait(mutex);
...
critical section
...
wait(mutex);

```

In this case, a deadlock will occur.

- Suppose that a process omits the `wait(mutex)`, or the `signal(mutex)`, or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem. Similar problems may arise in the other synchronization models that we discussed in Section 6.6.

To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the **monitor** type.

6.7.1 Usage

A type, or abstract data type, encapsulates private data with public methods to operate on that data. A monitor type presents a set of programmer-defined operations that are provided mutual exclusion within the monitor. The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables. The syntax of a monitor is shown in Figure 6.16. The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.

```

monitor monitor name
{
    // shared variable declarations

    procedure P1 ( . . . ) {
        . . .
    }

    procedure P2 ( . . . ) {
        . . .
    }

    .
    .
    .
    procedure Pn ( . . . ) {
        . . .
    }

    initialization code ( . . . ) {
        . . .
    }
}

```

Figure 6.16 Syntax of a monitor.

The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly (Figure 6.17). However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*:

```
condition x, y;
```

The only operations that can be invoked on a condition variable are `wait()` and `signal()`. The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal();
```

The `x.signal()` operation resumes exactly one suspended process. If no process is suspended, then the `signal()` operation has no effect; that is, the state of `x` is the same as if the operation had never been executed (Figure 6.18). Contrast this operation with the `signal()` operation associated with semaphores, which always affects the state of the semaphore.

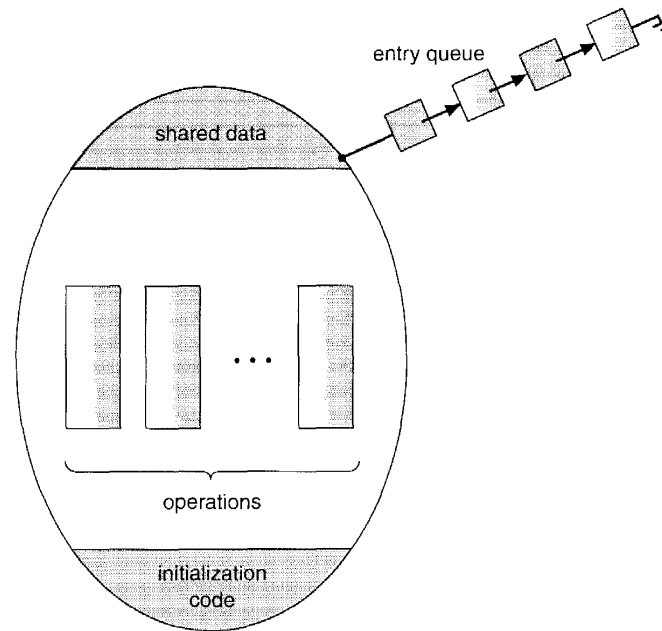


Figure 6.17 Schematic view of a monitor.

Now suppose that, when the `x.signal()` operation is invoked by a process P , there is a suspended process Q associated with condition x . Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that both processes can conceptually continue with their execution. Two possibilities exist:

1. **Signal and wait.** P either waits until Q leaves the monitor or waits for another condition.
2. **Signal and continue.** Q either waits until P leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option. On the one hand, since P was already executing in the monitor, the *signal-and-continue* method seems more reasonable. On the other hand, if we allow thread P to continue, then by the time Q is resumed, the logical condition for which Q was waiting may no longer hold. A compromise between these two choices was adopted in the language Concurrent Pascal. When thread P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed.

6.7.2 Dining-Philosophers Solution Using Monitors

We now illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To

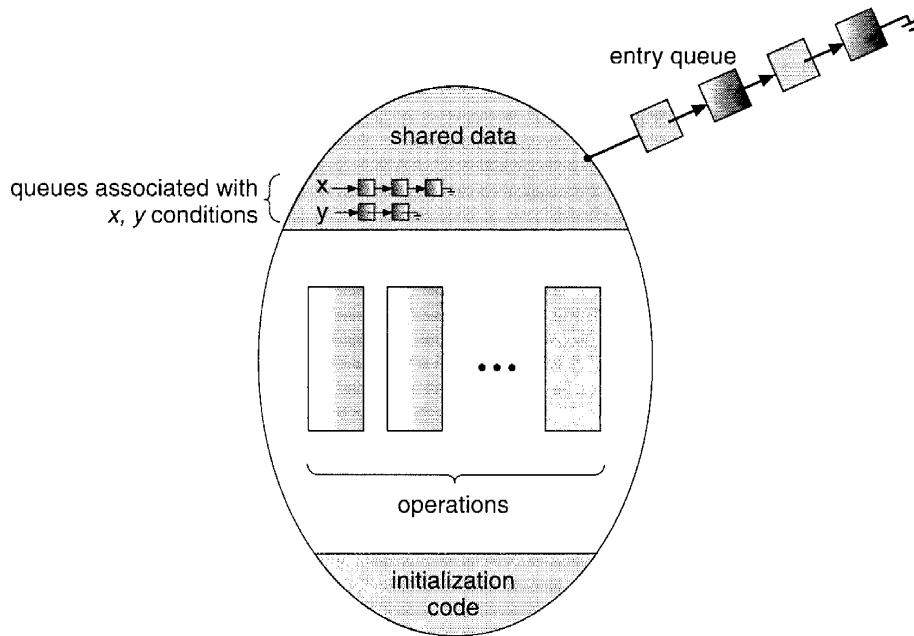


Figure 6.18 Monitor with condition variables.

code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum {thinking, hungry, eating} state[5];
```

Philosopher i can set the variable $state[i] = \text{eating}$ only if her two neighbors are not eating: $(state[(i+4) \% 5] \neq \text{eating})$ and $(state[(i+1) \% 5] \neq \text{eating})$.

We also need to declare

```
condition self[5];
```

where philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor dp , whose definition is shown in Figure 6.19. Each philosopher, before starting to eat, must invoke the operation $\text{pickup}()$. This may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the $\text{putdown}()$ operation. Thus, philosopher i must invoke the operations $\text{pickup}()$ and $\text{putdown}()$ in the following sequence:

```
dp.pickup(i);
...
eat
...
dp.putdown(i);
```

```

monitor dp
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)) {
            state[i] = EATING;
            self[i].signal();
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}

```

Figure 6.19 A monitor solution to the dining-philosopher problem.

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death. We do not present a solution to this problem but rather leave it as an exercise for you.

6.7.3 Implementing a Monitor Using Semaphores

We now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a semaphore `mutex` (initialized to 1) is provided. A process must execute `wait(mutex)` before entering the monitor and must execute `signal(mutex)` after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, `next`, is introduced, initialized to 0, on which the signaling processes may suspend themselves. An integer variable

`next_count` is also provided to count the number of processes suspended on `next`. Thus, each external procedure `F` is replaced by

```
wait(mutex);
...
body of F
...
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented. For each condition `x`, we introduce a semaphore `x_sem` and an integer variable `x_count`, both initialized to 0. The operation `x.wait()` can now be implemented as

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

The operation `x.signal()` can be implemented as

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

This implementation is applicable to the definitions of monitors given by both Hoare and Brinch-Hansen. In some cases, however, the generality of the implementation is unnecessary, and a significant improvement in efficiency is possible. We leave this problem to you in Exercise 6.17.

6.7.4 Resuming Processes Within a Monitor

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition `x`, and an `x.signal()` operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use an FCFS ordering, so that the process waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. For this purpose, the **conditional-wait** construct can be used; it has the form

```
x.wait(c);
```

```

monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }

    void release() {
        busy = FALSE;
        x.signal();
    }

    initialization_code() {
        busy = FALSE;
    }
}

```

Figure 6.20 A monitor to allocate a single resource.

where c is an integer expression that is evaluated when the `wait()` operation is executed. The value of c , which is called a **priority number**, is then stored with the name of the process that is suspended. When `x.signal()` is executed, the process with the smallest associated priority number is resumed next.

To illustrate this new mechanism, we consider the `ResourceAllocator` monitor shown in Figure 6.20, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation request. A process that needs to access the resource in question must observe the following sequence:

```

R.acquire(t);
...
access the resource;
...
R.release();

```

where R is an instance of type `ResourceAllocator`.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.

- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

The same difficulties are encountered with the use of semaphores, and these difficulties are similar in nature to those that encouraged us to develop the monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores. Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the current problem is to include the resource-access operations within the `ResourceAllocator` monitor. However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the `ResourceAllocator` monitor and its managed resource. We must check two conditions to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur and that the scheduling algorithm will not be defeated.

Although this inspection may be possible for a small, static system, it is not reasonable for a large system or a dynamic system. This access-control problem can be solved only by additional mechanisms that will be described in Chapter 14.

Many programming languages have incorporated the idea of the monitor as described in this section, including Concurrent Pascal, Mesa, C# (pronounced *C-sharp*), and Java. Other languages—such as Erlang—provide some type of concurrency support using a similar mechanism.

6.8 Synchronization Examples

We next describe the synchronization mechanisms provided by the Solaris, Windows XP, and Linux operating systems, as well as the Pthreads API. We have chosen these three systems because they provide good examples of different approaches for synchronizing the kernel, and we have included the Pthreads API because it is widely used for thread creation and synchronization by developers on UNIX and Linux systems. As you will see in this section, the synchronization methods available in these differing systems vary in subtle and significant ways.

6.8.1 Synchronization in Solaris

To control access to critical sections, Solaris provides adaptive mutexes, condition variables, semaphores, reader–writer locks, and turnstiles. Solaris implements semaphores and condition variables essentially as they are presented

JAVA MONITORS

Java provides a monitor-like concurrency mechanism for thread synchronization. Every object in Java has associated with it a single lock. When a method is declared to be `synchronized`, calling the method requires owning the lock for the object. We declare a `synchronized` method by placing the `synchronized` keyword in the method definition. The following defines the `safeMethod()` as `synchronized`, for example:

```
public class SimpleClass {
    public synchronized void safeMethod() {
        /* Implementation of safeMethod() */
        . . .
    }
}
```

Next, assume we create an object instance of `SimpleClass`, such as:

```
SimpleClass sc = new SimpleClass();
```

Invoking the `sc.safeMethod()` method requires owning the lock on the object instance `sc`. If the lock is already owned by another thread, the thread calling the `synchronized` method blocks and is placed in the **entry set** for the object's lock. The entry set represents the set of threads waiting for the lock to become available. If the lock is available when a `synchronized` method is called, the calling thread becomes the owner of the object's lock and can enter the method. The lock is released when the thread exits the method; a thread from the entry set is then selected as the new owner of the lock.

Java also provides `wait()` and `notify()` methods, which are similar in function to the `wait()` and `signal()` statements for a monitor. Release 1.5 of the Java Virtual Machine provides API support for semaphores, condition variables, and mutex locks (among other concurrency mechanisms) in the `java.util.concurrent` package.

in Sections 6.5 and 6.7. In this section, we describe adaptive mutexes, reader-writer locks, and turnstiles.

An **adaptive mutex** protects access to every critical data item. On a multiprocessor system, an adaptive mutex starts as a standard semaphore implemented as a spinlock. If the data are locked and therefore already in use, the adaptive mutex does one of two things. If the lock is held by a thread that is currently running on another CPU, the thread spins while waiting for the lock to become available, because the thread holding the lock is likely to finish soon. If the thread holding the lock is not currently in run state, the thread blocks, going to sleep until it is awakened by the release of the lock. It is put to sleep so that it will not spin while waiting, since the lock will not be freed very soon. A lock held by a sleeping thread is likely to be in this category. On a single-processor system, the thread holding the lock is never running if the

lock is being tested by another thread, because only one thread can run at a time. Therefore, on this type of system, threads always sleep rather than spin if they encounter a lock.

Solaris uses the adaptive-mutex method to protect only data that are accessed by short code segments. That is, a mutex is used if a lock will be held for less than a few hundred instructions. If the code segment is longer than that, spin waiting will be exceedingly inefficient. For these longer code segments, condition variables and semaphores are used. If the desired lock is already held, the thread issues a wait and sleeps. When a thread frees the lock, it issues a signal to the next sleeping thread in the queue. The extra cost of putting a thread to sleep and waking it, and of the associated context switches, is less than the cost of wasting several hundred instructions waiting in a spinlock.

Reader–writer locks are used to protect data that are accessed frequently but are usually accessed in a read-only manner. In these circumstances, reader–writer locks are more efficient than semaphores, because multiple threads can read data concurrently, whereas semaphores always serialize access to the data. Reader–writer locks are relatively expensive to implement, so again they are used on only long sections of code.

Solaris uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or a reader–writer lock. A **turnstile** is a queue structure containing threads blocked on a lock. For example, if one thread currently owns the lock for a synchronized object, all other threads trying to acquire the lock will block and enter the turnstile for that lock. When the lock is released, the kernel selects a thread from the turnstile as the next owner of the lock. Each synchronized object with at least one thread blocked on the object's lock requires a separate turnstile. However, rather than associating a turnstile with each synchronized object, Solaris gives each kernel thread its own turnstile. Because a thread can be blocked only on one object at a time, this is more efficient than having a turnstile per object.

The turnstile for the first thread to block on a synchronized object becomes the turnstile for the object itself. Subsequent threads blocking on the lock will be added to this turnstile. When the initial thread ultimately releases the lock, it gains a new turnstile from a list of free turnstiles maintained by the kernel. To prevent a **priority inversion**, turnstiles are organized according to a **priority-inheritance protocol** (Section 19.5). This means that if a lower-priority thread currently holds a lock that a higher-priority thread is blocked on, the thread with the lower priority will temporarily inherit the priority of the higher-priority thread. Upon releasing the lock, the thread will revert to its original priority.

Note that the locking mechanisms used by the kernel are implemented for user-level threads as well, so the same types of locks are available inside and outside the kernel. A crucial implementation difference is the priority-inheritance protocol. Kernel-locking routines adhere to the kernel priority-inheritance methods used by the scheduler, as described in Section 19.5; user-level thread-locking mechanisms do not provide this functionality.

To optimize Solaris performance, developers have refined and fine-tuned the locking methods. Because locks are used frequently and typically are used for crucial kernel functions, tuning their implementation and use can produce great performance gains.

6.8.2 Synchronization in Windows XP

The Windows XP operating system is a multithreaded kernel that provides support for real-time applications and multiple processors. When the Windows XP kernel accesses a global resource on a uniprocessor system, it temporarily masks interrupts for all interrupt handlers that may also access the global resource. On a multiprocessor system, Windows XP protects access to global resources using spinlocks. Just as in Solaris, the kernel uses spinlocks only to protect short code segments. Furthermore, for reasons of efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock.

For thread synchronization outside the kernel, Windows XP provides **dispatcher objects**. Using a dispatcher object, threads synchronize according to several different mechanisms, including mutexes, semaphores, events, and timers. The system protects shared data by requiring a thread to gain ownership of a mutex to access the data and to release ownership when it is finished. Semaphores behave as described in Section 6.5. **Events** are similar to condition variables; that is, they may notify a waiting thread when a desired condition occurs. Finally, timers are used to notify one (or more than one) thread that a specified amount of time has expired.

Dispatcher objects may be in either a signaled state or a nonsignaled state. A **signaled state** indicates that an object is available and a thread will not block when acquiring the object. A **nonsignaled state** indicates that an object is not available and a thread will block when attempting to acquire the object. We illustrate the state transitions of a mutex lock dispatcher object in Figure 6.21.

A relationship exists between the state of a dispatcher object and the state of a thread. When a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting, and the thread is placed in a waiting queue for that object. When the state for the dispatcher object moves to signaled, the kernel checks whether any threads are waiting on the object. If so, the kernel moves one thread—or possibly more threads—from the waiting state to the ready state, where they can resume executing. The number of threads the kernel selects from the waiting queue depends on the type of dispatcher object it is waiting on. The kernel will select only one thread from the waiting queue for a mutex, since a mutex object may be “owned” by only a single thread. For an event object, the kernel will select all threads that are waiting for the event.

We can use a mutex lock as an illustration of dispatcher objects and thread states. If a thread tries to acquire a mutex dispatcher object that is in a nonsignaled state, that thread will be suspended and placed in a waiting queue for the mutex object. When the mutex moves to the signaled state (because another thread has released the lock on the mutex), the thread waiting at the

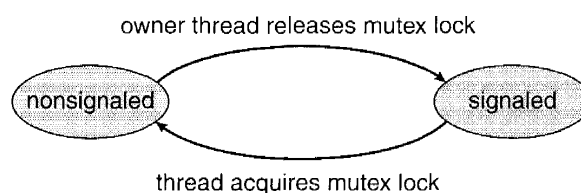


Figure 6.21 Mutex dispatcher object.

front of the queue will be moved from the waiting state to the ready state and will acquire the mutex lock.

We provide a programming project at the end of this chapter that uses mutex locks and semaphores in the Win32 API.

6.8.3 Synchronization in Linux

Prior to version 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run. Now, however, the Linux kernel is fully preemptive, so a task can be preempted when it is running in the kernel.

The Linux kernel provides spinlocks and semaphores (as well as reader–writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that the spinlock is held only for short durations. On single-processor machines, spinlocks are inappropriate for use and are replaced by enabling and disabling kernel preemption. That is, on single-processor machines, rather than holding a spinlock, the kernel disables kernel preemption; and rather than releasing the spinlock, it enables kernel preemption. This is summarized below:

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Linux uses an interesting approach to disable and enable kernel preemption. It provides two simple system calls—`preempt_disable()` and `preempt_enable()`—for disabling and enabling kernel preemption. In addition, however, the kernel is not preemptible if a kernel-mode task is holding a lock. To enforce this, each task in the system has a `thread_info` structure containing a counter, `preempt_count`, to indicate the number of locks being held by the task. When a lock is acquired, `preempt_count` is incremented. It is decremented when a lock is released. If the value of `preempt_count` for the task currently running is greater than zero, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is zero, the kernel can safely be interrupted (assuming there are no outstanding calls to `preempt_disable()`).

Spinlocks—along with enabling and disabling kernel preemption—are used in the kernel only when a lock (or disabling kernel preemption) is held for a short duration. When a lock must be held for a longer period, semaphores are appropriate for use.

6.8.4 Synchronization in Pthreads

The Pthreads API provides mutex locks, condition variables, and read–write locks for thread synchronization. This API is available for programmers and is not part of any particular kernel. Mutex locks represent the fundamental synchronization technique used with Pthreads. A mutex lock is used to protect critical sections of code—that is, a thread acquires the lock before entering a critical section and releases it upon exiting the critical section. Condition variables in Pthreads behave much as described in Section 6.7. Read–write

locks behave similarly to the locking mechanism described in Section 6.6.2. Many systems that implement Pthreads also provide semaphores, although they are not part of the Pthreads standard and instead belong to the POSIX SEM extension. Other extensions to the Pthreads API include spinlocks, although not all extensions are considered portable from one implementation to another. We provide a programming project at the end of this chapter that uses Pthreads mutex locks and semaphores.

6.9 Atomic Transactions

The mutual exclusion of critical sections ensures that the critical sections are executed atomically. That is, if two critical sections are executed concurrently, the result is equivalent to their sequential execution in some unknown order. Although this property is useful in many application domains, in many cases we would like to make sure that a critical section forms a single logical unit of work that either is performed in its entirety or is not performed at all. An example is funds transfer, in which one account is debited and another is credited. Clearly, it is essential for data consistency either that both the credit and debit occur or that neither occur.

Consistency of data, along with storage and retrieval of data, is a concern often associated with **database systems**. Recently, there has been an upsurge of interest in using database-systems techniques in operating systems. Operating systems can be viewed as manipulators of data; as such, they can benefit from the advanced techniques and models available from database research. For instance, many of the ad hoc techniques used in operating systems to manage files could be more flexible and powerful if more formal database methods were used in their place. In Sections 6.9.2 to 6.9.4, we describe some of these database techniques and explain how they can be used by operating systems. First, however, we deal with the general issue of transaction atomicity. It is this property that the database techniques are meant to address.

6.9.1 System Model

A collection of instructions (or operations) that performs a single logical function is called a **transaction**. A major issue in processing transactions is the preservation of atomicity despite the possibility of failures within the computer system.

We can think of a transaction as a program unit that accesses and perhaps updates various data items that reside on a disk within some files. From our point of view, such a transaction is simply a sequence of read and write operations terminated by either a **commit** operation or an **abort** operation. A **commit** operation signifies that the transaction has terminated its execution successfully, whereas an **abort** operation signifies that the transaction has ended its normal execution due to some logical error or a system failure. If a terminated transaction has completed its execution successfully, it is **committed**; otherwise, it is **aborted**.

Since an aborted transaction may already have modified the data that it has accessed, the state of these data may not be the same as it would have been if the transaction had executed atomically. So that atomicity is ensured,

an aborted transaction must have no effect on the state of the data that it has already modified. Thus, the state of the data accessed by an aborted transaction must be restored to what it was just before the transaction started executing. We say that such a transaction has been **rolled back**. It is part of the responsibility of the system to ensure this property.

To determine how the system should ensure atomicity, we need first to identify the properties of devices used for storing the various data accessed by the transactions. Various types of storage media are distinguished by their relative speed, capacity, and resilience to failure.

- **Volatile storage.** Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself and because it is possible to access directly any data item in volatile storage.
- **Nonvolatile storage.** Information residing in nonvolatile storage usually survives system crashes. Examples of media for such storage are disks and magnetic tapes. Disks are more reliable than main memory but less reliable than magnetic tapes. Both disks and tapes, however, are subject to failure, which may result in loss of information. Currently, nonvolatile storage is slower than volatile storage by several orders of magnitude, because disk and tape devices are electromechanical and require physical motion to access data.
- **Stable storage.** Information residing in stable storage is *never* lost (*never* should be taken with a grain of salt, since theoretically such absolutes cannot be guaranteed). To implement an approximation of such storage, we need to replicate information in several nonvolatile storage caches (usually disk) with independent failure modes and to update the information in a controlled manner (Section 12.8).

Here, we are concerned only with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.

6.9.2 Log-Based Recovery

One way to ensure atomicity is to record, on stable storage, information describing all the modifications made by the transaction to the various data it accesses. The most widely used method for achieving this form of recording is **write-ahead logging**. Here, the system maintains, on stable storage, a data structure called the **log**. Each log record describes a single operation of a transaction write and has the following fields:

- **Transaction name.** The unique name of the transaction that performed the write operation
- **Data item name.** The unique name of the data item written
- **Old value.** The value of the data item prior to the write operation
- **New value.** The value that the data item will have after the write

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.

Before a transaction T_i starts its execution, the record $\langle T_i \text{ starts} \rangle$ is written to the log. During its execution, any write operation by T_i is *preceded* by the writing of the appropriate new record to the log. When T_i commits, the record $\langle T_i \text{ commits} \rangle$ is written to the log.

Because the information in the log is used in reconstructing the state of the data items accessed by the various transactions, we cannot allow the actual update to a data item to take place before the corresponding log record is written out to stable storage. We therefore require that, prior to execution of a $\text{write}(X)$ operation, the log records corresponding to X be written onto stable storage.

Note the performance penalty inherent in this system. Two physical writes are required for every logical write requested. Also, more storage is needed, both for the data themselves and for the log recording the changes. In cases where the data are extremely important and fast failure recovery is necessary, the price is worth the functionality.

Using the log, the system can handle any failure that does not result in the loss of information on nonvolatile storage. The recovery algorithm uses two procedures:

- $\text{undo}(T_i)$, which restores the value of all data updated by transaction T_i to the old values
- $\text{redo}(T_i)$, which sets the value of all data updated by transaction T_i to the new values

The set of data updated by T_i and their respective old and new values can be found in the log.

The undo and redo operations must be idempotent (that is, multiple executions must have the same result as does one execution) to guarantee correct behavior, even if a failure occurs during the recovery process.

If a transaction T_i aborts, then we can restore the state of the data that it has updated by simply executing $\text{undo}(T_i)$. If a system failure occurs, we restore the state of all updated data by consulting the log to determine which transactions need to be redone and which need to be undone. This classification of transactions is accomplished as follows:

- Transaction T_i needs to be undone if the log contains the $\langle T_i \text{ starts} \rangle$ record but does not contain the $\langle T_i \text{ commits} \rangle$ record.
- Transaction T_i needs to be redone if the log contains both the $\langle T_i \text{ starts} \rangle$ and the $\langle T_i \text{ commits} \rangle$ records.

6.9.3 Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to make these determinations. There are two major drawbacks to this approach:

1. The searching process is time consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already actually updated the data that the log says they need to modify. Although redoing the data modifications will cause no harm (due to idempotency), it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce the concept of **checkpoints**. During execution, the system maintains the write-ahead log. In addition, the system periodically performs checkpoints that require the following sequence of actions to take place:

1. Output all log records currently residing in volatile storage (usually main memory) onto stable storage.
2. Output all modified data residing in volatile storage to the stable storage.
3. Output a log record `<checkpoint>` onto stable storage.

The presence of a `<checkpoint>` record in the log allows the system to streamline its recovery procedure. Consider a transaction T_i that committed prior to the checkpoint. The `<Ti commits>` record appears in the log before the `<checkpoint>` record. Any modifications made by T_i must have been written to stable storage either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on T_i .

This observation allows us to refine our previous recovery algorithm. After a failure has occurred, the recovery routine examines the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place. It finds such a transaction by searching the log backward to find the first `<checkpoint>` record, and then finding the subsequent `<Ti start>` record.

Once transaction T_i has been identified, the redo and undo operations need be applied only to transaction T_i and all transactions T_j that started executing after transaction T_i . We'll call these transactions set T . The remainder of the log can thus be ignored. The recovery operations that are required are as follows:

- For all transactions T_k in T such that the record `<Tk commits>` appears in the log, execute `redo(Tk)`.
- For all transactions T_k in T that have no `<Tk commits>` record in the log, execute `undo(Tk)`.

6.9.4 Concurrent Atomic Transactions

We have been considering an environment in which only one transaction can be executing at a time. We now turn to the case where multiple transactions are active simultaneously. Because each transaction is atomic, the concurrent execution of transactions must be equivalent to the case where these transactions are executed serially in some arbitrary order. This property, called **serializability**, can be maintained by simply executing each transaction within

a critical section. That is, all transactions share a common semaphore *mutex*, which is initialized to 1. When a transaction starts executing, its first action is to execute `wait(mutex)`. After the transaction either commits or aborts, it executes `signal(mutex)`.

Although this scheme ensures the atomicity of all concurrently executing transactions, it is nevertheless too restrictive. As we shall see, in many cases we can allow transactions to overlap their execution while maintaining serializability. A number of different **concurrency-control algorithms** ensure serializability. These algorithms are described below.

6.9.4.1 Serializability

Consider a system with two data items, *A* and *B*, that are both read and written by two transactions, T_0 and T_1 . Suppose that these transactions are executed atomically in the order T_0 followed by T_1 . This execution sequence, which is called a **schedule**, is represented in Figure 6.22. In schedule 1 of Figure 6.22, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T_0 appearing in the left column and instructions of T_1 appearing in the right column.

A schedule in which each transaction is executed atomically is called a **serial schedule**. A serial schedule consists of a sequence of instructions from various transactions wherein the instructions belonging to a particular transaction appear together. Thus, for a set of n transactions, there exist $n!$ different valid serial schedules. Each serial schedule is correct, because it is equivalent to the atomic execution of the various participating transactions in some arbitrary order.

If we allow the two transactions to overlap their execution, then the resulting schedule is no longer serial. A **nonserial schedule** does not necessarily imply an incorrect execution (that is, an execution that is not equivalent to one represented by a serial schedule). To see that this is the case, we need to define the notion of **conflicting operations**.

Consider a schedule S in which there are two consecutive operations O_i and O_j of transactions T_i and T_j , respectively. We say that O_i and O_j *conflict* if they access the same data item and at least one of them is a write operation. To illustrate the concept of conflicting operations, we consider the nonserial

T_0	T_1
read(<i>A</i>)	
write(<i>A</i>)	
read(<i>B</i>)	
write(<i>B</i>)	
	read(<i>A</i>)
	write(<i>A</i>)
	read(<i>B</i>)
	write(<i>B</i>)

Figure 6.22 Schedule 1: A serial schedule in which T_0 is followed by T_1 .

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 6.23 Schedule 2: A concurrent serializable schedule.

schedule 2 of Figure 6.23. The `write(A)` operation of T_0 conflicts with the `read(A)` operation of T_1 . However, the `write(A)` operation of T_1 does not conflict with the `read(B)` operation of T_0 , because the two operations access different data items.

Let O_i and O_j be consecutive operations of a schedule S . If O_i and O_j are operations of different transactions and O_i and O_j do not conflict, then we can swap the order of O_i and O_j to produce a new schedule S' . We expect S to be equivalent to S' , as all operations appear in the same order in both schedules, except for O_i and O_j , whose order does not matter.

We can illustrate the swapping idea by considering again schedule 2 of Figure 6.23. As the `write(A)` operation of T_1 does not conflict with the `read(B)` operation of T_0 , we can swap these operations to generate an equivalent schedule. Regardless of the initial system state, both schedules produce the same final system state. Continuing with this procedure of swapping nonconflicting operations, we get:

- Swap the `read(B)` operation of T_0 with the `read(A)` operation of T_1 .
- Swap the `write(B)` operation of T_0 with the `write(A)` operation of T_1 .
- Swap the `write(B)` operation of T_0 with the `read(A)` operation of T_1 .

The final result of these swaps is schedule 1 in Figure 6.22, which is a serial schedule. Thus, we have shown that schedule 2 is equivalent to a serial schedule. This result implies that, regardless of the initial system state, schedule 2 will produce the same final state as will some serial schedule.

If a schedule S can be transformed into a serial schedule S' by a series of swaps of nonconflicting operations, we say that a schedule S is **conflict serializable**. Thus, schedule 2 is conflict serializable, because it can be transformed into the serial schedule 1.

6.9.4.2 Locking Protocol

One way to ensure serializability is to associate with each data item a lock and to require that each transaction follow a **locking protocol** that governs how locks are acquired and released. There are various modes in which a data item can be locked. In this section, we restrict our attention to two modes:

- **Shared.** If a transaction T_i has obtained a shared-mode lock (denoted by S) on data item Q , then T_i can read this item but cannot write Q .
- **Exclusive.** If a transaction T_i has obtained an exclusive-mode lock (denoted by X) on data item Q , then T_i can both read and write Q .

We require that every transaction request a lock in an appropriate mode on data item Q , depending on the type of operations it will perform on Q .

To access data item Q , transaction T_i must first lock Q in the appropriate mode. If Q is not currently locked, then the lock is granted, and T_i can now access it. However, if the data item Q is currently locked by some other transaction, then T_i may have to wait. More specifically, suppose that T_i requests an exclusive lock on Q . In this case, T_i must wait until the lock on Q is released. If T_i requests a shared lock on Q , then T_i must wait if Q is locked in exclusive mode. Otherwise, it can obtain the lock and access Q . Notice that this scheme is quite similar to the readers–writers algorithm discussed in Section 6.6.2.

A transaction may unlock a data item that it locked at an earlier point. It must, however, hold a lock on a data item as long as it accesses that item. Moreover, it is not always desirable for a transaction to unlock a data item immediately after its last access of that data item, because serializability may not be ensured.

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

- **Growing phase.** A transaction may obtain locks but may not release any lock.
- **Shrinking phase.** A transaction may release locks but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and no more lock requests can be issued.

The two-phase locking protocol ensures conflict serializability (Exercise 6.25). It does not, however, ensure freedom from deadlock. In addition, it is possible that, for a given set of transactions, there are conflict-serializable schedules that cannot be obtained by use of the two-phase locking protocol. However, to improve performance over two-phase locking, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data.

6.9.4.3 Timestamp-Based Protocols

In the locking protocols described above, the order followed by pairs of conflicting transactions is determined at execution time by the first lock that both request and that involves incompatible modes. Another method for determining the serializability order is to select an order in advance. The most common method for doing so is to use a **timestamp** ordering scheme.

With each transaction T_i in the system, we associate a unique fixed timestamp, denoted by $TS(T_i)$. This timestamp is assigned by the system

before the transaction T_i starts execution. If a transaction T_i has been assigned timestamp $TS(T_i)$, and later a new transaction T_j enters the system, then $TS(T_i) < TS(T_j)$. There are two simple methods for implementing this scheme:

- Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system. This method will not work for transactions that occur on separate systems or for processors that do not share a clock.
- Use a logical counter as the timestamp; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system. The counter is incremented after a new timestamp is assigned.

The timestamps of the transactions determine the serializability order. Thus, if $TS(T_i) < TS(T_j)$, then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j .

To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp**(Q) denotes the largest timestamp of any transaction that successfully executed $\text{write}(Q)$.
- **R-timestamp**(Q) denotes the largest timestamp of any transaction that successfully executed $\text{read}(Q)$.

These timestamps are updated whenever a new $\text{read}(Q)$ or $\text{write}(Q)$ instruction is executed.

The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

- Suppose that transaction T_i issues $\text{read}(Q)$:
 - If $TS(T_i) < \text{W-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and T_i is rolled back.
 - If $TS(T_i) \geq \text{W-timestamp}(Q)$, then the read operation is executed, and $\text{R-timestamp}(Q)$ is set to the maximum of $\text{R-timestamp}(Q)$ and $TS(T_i)$.
- Suppose that transaction T_i issues $\text{write}(Q)$:
 - If $TS(T_i) < \text{R-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously and T_i assumed that this value would never be produced. Hence, the write operation is rejected, and T_i is rolled back.
 - If $TS(T_i) < \text{W-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q . Hence, this write operation is rejected, and T_i is rolled back.
 - Otherwise, the write operation is executed.

A transaction T_i that is rolled back as a result of the issuing of either a read or write operation is assigned a new timestamp and is restarted.

T_2	T_3
read(B)	read(B)
	write(B)
read(A)	read(A)
	write(A)

Figure 6.24 Schedule 3: A schedule possible under the timestamp protocol.

To illustrate this protocol, consider schedule 3 of Figure 6.24, which includes transactions T_2 and T_3 . We assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3, $TS(T_2) < TS(T_3)$, and the schedule is possible under the timestamp protocol.

This execution can also be produced by the two-phase locking protocol. However, some schedules are possible under the two-phase locking protocol but not under the timestamp protocol, and vice versa.

The timestamp protocol ensures conflict serializability. This capability follows from the fact that conflicting operations are processed in timestamp order. The protocol also ensures freedom from deadlock, because no transaction ever waits.

6.10 Summary

Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided. One solution is to ensure that a critical section of code is in use by only one process or thread at a time. Different algorithms exist for solving the critical-section problem, with the assumption that only storage interlock is available.

The main disadvantage of these user-coded solutions is that they all require busy waiting. Semaphores overcome this difficulty. Semaphores can be used to solve various synchronization problems and can be implemented efficiently, especially if hardware support for atomic operations is available.

Various synchronization problems (such as the bounded-buffer problem, the readers–writers problem, and the dining-philosophers problem) are important mainly because they are examples of a large class of concurrency-control problems. These problems are used to test nearly every newly proposed synchronization scheme.

The operating system must provide the means to guard against timing errors. Several language constructs have been proposed to deal with these problems. Monitors provide the synchronization mechanism for sharing abstract data types. A condition variable provides a method by which a monitor procedure can block its execution until it is signaled to continue.

Operating systems also provide support for synchronization. For example, Solaris, Windows XP, and Linux provide mechanisms such as semaphores, mutexes, spinlocks, and condition variables to control access to shared data. The Pthreads API provides support for mutexes and condition variables.

A transaction is a program unit that must be executed atomically; that is, either all the operations associated with it are executed to completion, or none are performed. To ensure atomicity despite system failure, we can use a write-ahead log. All updates are recorded on the log, which is kept in stable storage. If a system crash occurs, the information in the log is used in restoring the state of the updated data items, which is accomplished by use of the undo and redo operations. To reduce the overhead in searching the log after a system failure has occurred, we can use a checkpoint scheme.

To ensure serializability when the execution of several transactions overlaps, we must use a concurrency-control scheme. Various concurrency-control schemes ensure serializability by delaying an operation or aborting the transaction that issued the operation. The most common ones are locking protocols and timestamp ordering schemes.

Exercises

- 6.1 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1 , share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i ($i == 0$ or 1) is shown in Figure 6.25; the other process is P_j ($j == 1$ or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
do {
    flag[i] = TRUE;

    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
                ; // do nothing
            flag[i] = TRUE;
        }
    }

    // critical section

    turn = j;
    flag[i] = FALSE;

    // remainder section
}while (TRUE);
```

Figure 6.25 The structure of process P_i in Dekker's algorithm.

```

do {
    while (TRUE) {
        flag[i] = want_in;
        j = turn;

        while (j != i) {
            if (flag[j] != idle) {
                j = turn;
            }
            else
                j = (j + 1) % n;
        }

        flag[i] = in_cs;
        j = 0;

        while ( (j < n) && (j == i || flag[j] != in_cs) )
            j++;

        if ( (j >= n) && (turn == i || flag[turn] == idle) )
            break;
    }

    // critical section

    j = (turn + 1) % n;

    while (flag[j] == idle)
        j = (j + 1) % n;

    turn = j;
    flag[i] = idle;

    // remainder section
}while (TRUE);

```

Figure 6.26 The structure of process P_i in Eisenberg and McGuire's algorithm.

- 6.2** The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of $n - 1$ turns was presented by Eisenberg and McGuire. The processes share the following variables:

```

enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;

```

All the elements of `flag` are initially `idle`; the initial value of `turn` is immaterial (between 0 and $n-1$). The structure of process P_i is shown in Figure 6.26. Prove that the algorithm satisfies all three requirements for the critical-section problem.

- 6.3 What is the meaning of the term *busy waiting*? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.
- 6.4 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.
- 6.5 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.
- 6.6 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.
- 6.7 Describe how the `Swap()` instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.
- 6.8 Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.
- 6.9 Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.
- 6.10 Show how to implement the `wait()` and `signal()` semaphore operations in multiprocessor environments using the `TestAndSet()` instruction. The solution should exhibit minimal busy waiting.
- 6.11 **The Sleeping-Barber Problem.** A barbershop consists of a waiting room with n chairs and a barber room with one barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.
- 6.12 Demonstrate that monitors and semaphores are equivalent insofar as they can be used to implement the same types of synchronization problems.
- 6.13 Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.
- 6.14 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 6.13 mainly suitable for small portions.
 - a. Explain why this is true.
 - b. Design a new scheme that is suitable for larger portions.
- 6.15 Discuss the tradeoff between fairness and throughput of operations in the readers–writers problem. Propose a method for solving the readers–writers problem without causing starvation.

- 6.16 How does the `signal()` operation associated with monitors differ from the corresponding operation defined for semaphores?
- 6.17 Suppose the `signal()` statement can appear only as the last statement in a monitor procedure. Suggest how the implementation described in Section 6.7 can be simplified.
- 6.18 Consider a system consisting of processes $P_1, P_2, \dots, P_n$, each of which has a unique priority number. Write a monitor that allocates three identical line printers to these processes, using the priority numbers for deciding the order of allocation.
- 6.19 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: The sum of all unique numbers associated with all the processes currently accessing the file must be less than n . Write a monitor to coordinate access to the file.
- 6.20 When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with the two different ways in which signaling can be performed?
- 6.21 Suppose we replace the `wait()` and `signal()` operations of monitors with a single construct `await(B)`, where B is a general Boolean expression that causes the process executing it to wait until B becomes true.
- Write a monitor using this scheme to implement the readers-writers problem.
 - Explain why, in general, this construct cannot be implemented efficiently.
 - What restrictions need to be put on the `await` statement so that it can be implemented efficiently? (Hint: Restrict the generality of B ; see Kessels [1977].)
- 6.22 Write a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a procedure *tick* in your monitor at regular intervals.
- 6.23 Why do Solaris, Linux, and Windows 2000 use spinlocks as a synchronization mechanism only on multiprocessor systems and not on single-processor systems?
- 6.24 In log-based systems that provide support for transactions, updates to data items cannot be performed before the corresponding entries are logged. Why is this restriction necessary?
- 6.25 Show that the two-phase locking protocol ensures conflict serializability.
- 6.26 What are the implications of assigning a new timestamp to a transaction that is rolled back? How does the system process transactions that were issued after the rolled-back transaction but that have timestamps smaller than the new timestamp of the rolled-back transaction?

- 6.27 Assume that a finite number of resources of a single resource type must be managed. Processes may ask for a number of these resources and —once finished— will return them. As an example, many commercial software packages provide a given number of **licenses**, indicating the number of applications that may run concurrently. When the application is started, the license count is decremented. When the application is terminated, the license count is incremented. If all licenses are in use, requests to start the application are denied. Such requests will only be granted when an existing license holder terminates the application and a license is returned.

The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the `decrease_count()` function:

```
/* decrease available_resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease_count(int count) {
    if (available_resources < count)
        return -1;
    else {
        available_resources -= count;

        return 0;
    }
}
```

When a process wants to return a number of resources, it calls the `increase_count()` function:

```
/* increase available_resources by count */
int increase_count(int count) {
    available_resources += count;

    return 0;
}
```

The preceding program segment produces a race condition. Do the following:

- Identify the data involved in the race condition.
- Identify the location (or locations) in the code where the race condition occurs.
- Using a semaphore, fix the race condition.

- 6.28 The `decrease_count()` function in the previous exercise currently returns 0 if sufficient resources are available and -1 otherwise. This leads to awkward programming for a process that wishes obtain a number of resources:

```
while (decrease_count(count) == -1)
    ;
```

Rewrite the resource-manager code segment using a monitor and condition variables so that the `decrease_count()` function suspends the process until sufficient resources are available. This will allow a process to invoke `decrease_count()` by simply calling

```
decrease_count(count);
```

The process will only return from this function call when sufficient resources are available.

Project: Producer–Consumer Problem

In Section 6.6.1, we present a semaphore-based solution to the producer–consumer problem using a bounded buffer. In this project, we will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 6.10 and 6.11. The solution presented in Section 6.6.1 uses three semaphores: `empty` and `full`, which count the number of empty and full slots in the buffer, and `mutex`, which is a binary (or mutual exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, standard counting semaphores will be used for `empty` and `full`, and, rather than a binary semaphore, a mutex lock will be used to represent `mutex`. The producer and consumer—running as separate threads—will move items to and from a buffer that is synchronized with these `empty`, `full`, and `mutex` structures. You can solve this problem using either Pthreads or the Win32 API.

The Buffer

Internally, the buffer will consist of a fixed-size array of type `buffer_item` (which will be defined using a `typedef`). The array of `buffer_item` objects will be manipulated as a circular queue. The definition of `buffer_item`, along with the size of the buffer, can be stored in a header file such as the following:

```
/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5
```

The buffer will be manipulated with two functions, `insert_item()` and `remove_item()`, which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears as:

```

#include <buffer.h>

/* the buffer */
buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {
    /* insert item into buffer
       return 0 if successful, otherwise
       return -1 indicating an error condition */
}

int remove_item(buffer_item *item) {
    /* remove an object from buffer
       placing it in item
       return 0 if successful, otherwise
       return -1 indicating an error condition */
}

```

The `insert_item()` and `remove_item()` functions will synchronize the producer and consumer using the algorithms outlined in Figures 6.10 and 6.11. The buffer will also require an initialization function that initializes the mutual-exclusion object `mutex` along with the empty and full semaphores.

The `main()` function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the `main()` function will sleep for a period of time and, upon awakening, will terminate the application. The `main()` function will be passed three parameters on the command line:

1. How long to sleep before terminating
2. The number of producer threads
3. The number of consumer threads

A skeleton for this function appears as:

```

#include <buffer.h>

int main(int argc, char *argv[]) {
    /* 1. Get command line arguments argv[1], argv[2], argv[3] */
    /* 2. Initialize buffer */
    /* 3. Create producer thread(s) */
    /* 4. Create consumer thread(s) */
    /* 5. Sleep */
    /* 6. Exit */
}

```

Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will

be produced using the `rand()` function, which produces random integers between 0 and `RAND_MAX`. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer. An outline of the producer and consumer threads appears as:

```
#include <stdlib.h> /* required for rand() */
#include <buffer.h>

void *producer(void *param) {
    buffer_item rand;

    while (TRUE) {
        /* sleep for a random period of time */
        sleep(...);
        /* generate a random number */
        rand = rand();
        printf("producer produced %f\n",rand);
        if (insert_item(rand))
            fprintf("report error condition");
    }
}

void *consumer(void *param) {
    buffer_item rand;

    while (TRUE) {
        /* sleep for a random period of time */
        sleep(...);
        if (remove_item(&rand))
            fprintf("report error condition");
        else
            printf("consumer consumed %f\n",rand);
    }
}
```

In the following sections, we first cover details specific to Pthreads and then describe details of the Win32 API.

Pthreads Thread Creation

Creating threads using the Pthreads API is discussed in Chapter 4. Please refer to that chapter for specific instructions regarding creation of the producer and consumer using Pthreads.

Pthreads Mutex Locks

The following code sample illustrates how mutex locks available in the Pthread API can be used to protect a critical section:

```

#include <pthread.h>
pthread_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex, NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/** critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);

```

Pthreads uses the `pthread_mutex_t` data type for mutex locks. A mutex is created with the `pthread_mutex_init(&mutex, NULL)` function, with the first parameter being a pointer to the mutex. By passing `NULL` as a second parameter, we initialize the mutex to its default attributes. The mutex is acquired and released with the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. If the mutex lock is unavailable when `pthread_mutex_lock()` is invoked, the calling thread is blocked until the owner invokes `pthread_mutex_unlock()`. All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.

Pthreads Semaphores

Pthreads provides two types of semaphores—named and unnamed. For this project, we use unnamed semaphores. The code below illustrates how a semaphore is created:

```

#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 5 */
sem_init(&sem, 0, 5);

```

The `sem_init()` creates and initializes a semaphore. This function is passed three parameters:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

In this example, by passing the flag 0, we are indicating that this semaphore can only be shared by threads belonging to the same process that created the semaphore. A nonzero value would allow other processes to access the semaphore as well. In this example, we initialize the semaphore to the value 5.

In Section 6.5, we described the classical `wait()` and `signal()` semaphore operations. Pthreads names the `wait()` and `signal()` operations `sem_wait()` and `sem_post()`, respectively. The code example below creates a binary semaphore `mutex` with an initial value of 1 and illustrates its use in protecting a critical section:

```
#include <semaphore.h>
sem_t sem mutex;

/* create the semaphore */
sem_init(&mutex, 0, 1);

/* acquire the semaphore */
sem_wait(&mutex);

/** critical section **/

/* release the semaphore */
sem_post(&mutex);
```

Win32

Details concerning thread creation using the Win32 API are available in Chapter 4. Please refer to that chapter for specific instructions.

Win32 Mutex Locks

Mutex locks are a type of dispatcher object, as described in Section 6.8.2. The following illustrates how to create a mutex lock using the `CreateMutex()` function:

```
#include <windows.h>

HANDLE Mutex;
Mutex = CreateMutex(NULL, FALSE, NULL);
```

The first parameter refers to a security attribute for the mutex lock. By setting this attribute to `NULL`, we are disallowing any children of the process creating this mutex lock to inherit the handle of the mutex. The second parameter indicates whether the creator of the mutex is the initial owner of the mutex lock. Passing a value of `FALSE` indicates that the thread creating the mutex is not the initial owner; we shall soon see how mutex locks are acquired. The third parameter allows naming of the mutex. However, because we provide a value of `NULL`, we do not name the mutex. If successful, `CreateMutex()` returns a `HANDLE` to the mutex lock; otherwise, it returns `NULL`.

In Section 6.8.2, we identified dispatcher objects as being either *signaled* or *nonsignaled*. A signaled object is available for ownership; once a dispatcher object (such as a mutex lock) is acquired, it moves to the nonsignaled state. When the object is released, it returns to signaled.

Mutex locks are acquired by invoking the `WaitForSingleObject()` function, passing the function the `HANDLE` to the lock and a flag indicating how long to wait. The following code demonstrates how the mutex lock created above can be acquired:

```
WaitForSingleObject(Mutex, INFINITE);
```

The parameter value `INFINITE` indicates that we will wait an infinite amount of time for the lock to become available. Other values could be used that would allow the calling thread to time out if the lock did not become available within a specified time. If the lock is in a signaled state, `WaitForSingleObject()` returns immediately, and the lock becomes nonsignaled. A lock is released (moves to the nonsignaled state) by invoking `ReleaseMutex()`, such as:

```
ReleaseMutex(Mutex);
```

Win32 Semaphores

Semaphores in the Win32 API are also dispatcher objects and thus use the same signaling mechanism as mutex locks. Semaphores are created as follows:

```
#include <windows.h>

HANDLE Sem;
Sem = CreateSemaphore(NULL, 1, 5, NULL);
```

The first and last parameters identify a security attribute and a name for the semaphore, similar to what was described for mutex locks. The second and third parameters indicate the initial value and maximum value of the semaphore. In this instance, the initial value of the semaphore is 1, and its maximum value is 5. If successful, `CreateSemaphore()` returns a `HANDLE` to the mutex lock; otherwise, it returns `NULL`.

Semaphores are acquired with the same `WaitForSingleObject()` function as mutex locks. We acquire the semaphore `Sem` created in this example by using the statement:

```
WaitForSingleObject(Semaphore, INFINITE);
```

If the value of the semaphore is > 0 , the semaphore is in the signaled state and thus is acquired by the calling thread. Otherwise, the calling thread blocks indefinitely—as we are specifying `INFINITE`—until the semaphore becomes signaled.

The equivalent of the `signal()` operation on Win32 semaphores is the `ReleaseSemaphore()` function. This function is passed three parameters: (1) the `HANDLE` of the semaphore, (2) the amount by which to increase the value of the semaphore, and (3) a pointer to the previous value of the semaphore. We can increase `Sem` by 1 using the following statement:

```
ReleaseSemaphore(Sem, 1, NULL);
```

Both `ReleaseSemaphore()` and `ReleaseMutex()` return 0 if successful and nonzero otherwise.

Bibliographical Notes

The mutual-exclusion problem was first discussed in a classic paper by Dijkstra [1965a]. Dekker's algorithm (Exercise 6.1)—the first correct software solution to the two-process mutual-exclusion problem—was developed by the Dutch mathematician T. Dekker. This algorithm also was discussed by Dijkstra [1965a]. A simpler solution to the two-process mutual-exclusion problem has since been presented by Peterson [1981] (Figure 6.2).

Dijkstra [1965b] presented the first solution to the mutual-exclusion problem for n processes. This solution, however does not have an upper bound on the amount of time a process must wait before it is allowed to enter the critical section. Knuth [1966] presented the first algorithm with a bound; his bound was 2^n turns. A refinement of Knuth's algorithm by deBruijn [1967] reduced the waiting time to n^2 turns, after which Eisenberg and McGuire [1972] (Exercise 6.4) succeeded in reducing the time to the lower bound of $n-1$ turns. Another algorithm that also requires $n-1$ turns but is easier to program and to understand, is the bakery algorithm, which was developed by Lamport [1974]. Burns [1978] developed the hardware-solution algorithm that satisfies the bounded-waiting requirement.

General discussions concerning the mutual-exclusion problem were offered by Lamport [1986] and Lamport [1991]. A collection of algorithms for mutual exclusion was given by Raynal [1986].

The semaphore concept was suggested by Dijkstra [1965a]. Patil [1971] examined the question of whether semaphores can solve all possible synchronization problems. Parnas [1975] discussed some of the flaws in Patil's arguments. Kosaraju [1973] followed up on Patil's work to produce a problem that cannot be solved by `wait()` and `signal()` operations. Lipton [1974] discussed the limitations of various synchronization primitives.

The classic process-coordination problems that we have described are paradigms for a large class of concurrency-control problems. The bounded-buffer problem, the dining-philosophers problem, and the sleeping-barber problem (Exercise 6.11) were suggested by Dijkstra [1965a] and Dijkstra [1971]. The cigarette-smokers problem (Exercise 6.8) was developed by Patil [1971]. The readers-writers problem was suggested by Courtois et al. [1971]. The issue of concurrent reading and writing was discussed by Lamport [1977]. The problem of synchronization of independent processes was discussed by Lamport [1976].

The critical-region concept was suggested by Hoare [1972] and by Brinch-Hansen [1972]. The monitor concept was developed by Brinch-Hansen [1973]. A complete description of the monitor was given by Hoare [1974]. Kessels [1977] proposed an extension to the monitor to allow automatic signaling. Experience obtained from the use of monitors in concurrent programs was discussed in Lamport and Redell [1979]. General discussions concerning concurrent programming were offered by Ben-Ari [1990] and Birrell [1989].

Optimizing the performance of locking primitives has been discussed in many works, such as Lamport [1987], Mellor-Crummey and Scott [1991], and Anderson [1990]. The use of shared objects that do not require the use of critical sections was discussed in Herlihy [1993], Bershad [1993], and Kopetz and Reisinger [1993]. Novel hardware instructions and their utility in implementing

synchronization primitives have been described in works such as Culler et al. [1998], Goodman et al. [1989], Barnes [1993], and Herlihy and Moss [1993].

Some details of the locking mechanisms used in Solaris were presented in Mauro and McDougall [2001]. Note that the locking mechanisms used by the kernel are implemented for user-level threads as well, so the same types of locks are available inside and outside the kernel. Details of Windows 2000 synchronization can be found in Solomon and Russinovich [2000].

The write-ahead log scheme was first introduced in System R by Gray et al. [1981]. The concept of serializability was formulated by Eswaran et al. [1976] in connection with their work on concurrency control for System R. The two-phase locking protocol was introduced by Eswaran et al. [1976]. The timestamp-based concurrency-control scheme was provided by Reed [1983]. An exposition of various timestamp-based concurrency-control algorithms was presented by Bernstein and Goodman [1980].