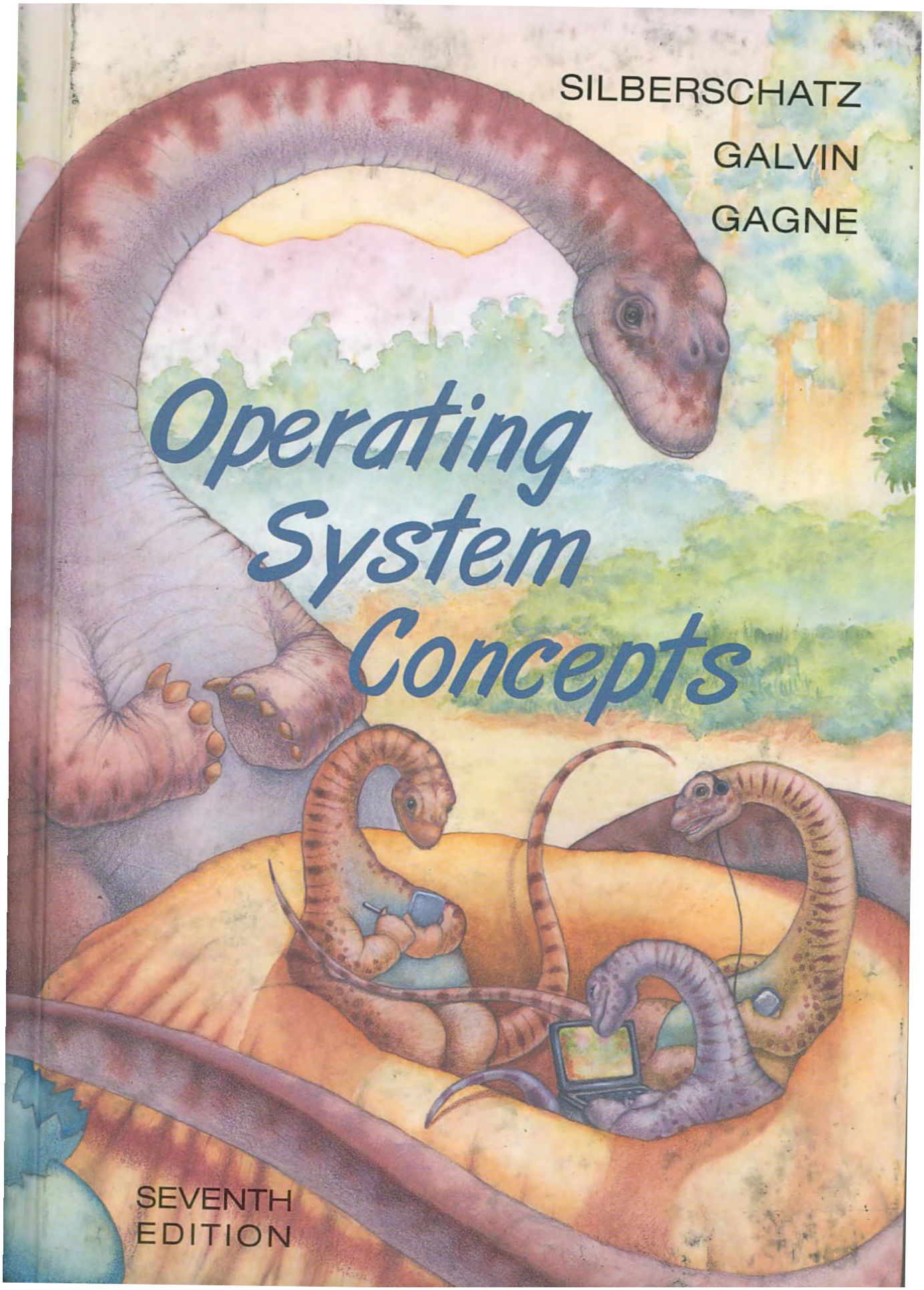




SAMSUNG 1025



**Abraham Silberschatz** is a Professor of Computer Science at Yale University. Prior to joining Yale, he was the Vice President of the Information Sciences Research Center at Bell Laboratories, Murray Hill, New Jersey. Prior to that, he held a chaired professorship in the Department of Computer Sciences at the University of Texas at Austin. His research interests include operating systems, database systems, real-time systems, storage systems, network management, and distributed systems.

In addition to his academic and industrial positions, Professor Silberschatz served as a member of the Biodiversity and Ecosystems Panel on President Clinton's Committee of Advisors on Science and Technology, as an advisor for the National Science Foundation, and as a consultant for several private industry companies.

Professor Silberschatz is an ACM Fellow and an IEEE Fellow. He received the 2002 IEEE Taylor L. Booth Education Award, the 1998 ACM Karl V. Karlstrom Outstanding Educator Award, the 1997 ACM SIGMOD Contribution Award, and the IEEE Computer Society Outstanding Paper award for the article "Capability Manager," which appeared in the *IEEE Transactions on Software Engineering*. His writings have appeared in numerous ACM and IEEE publications and other professional conferences and journals. He is a coauthor of the textbook *Database System Concepts*.

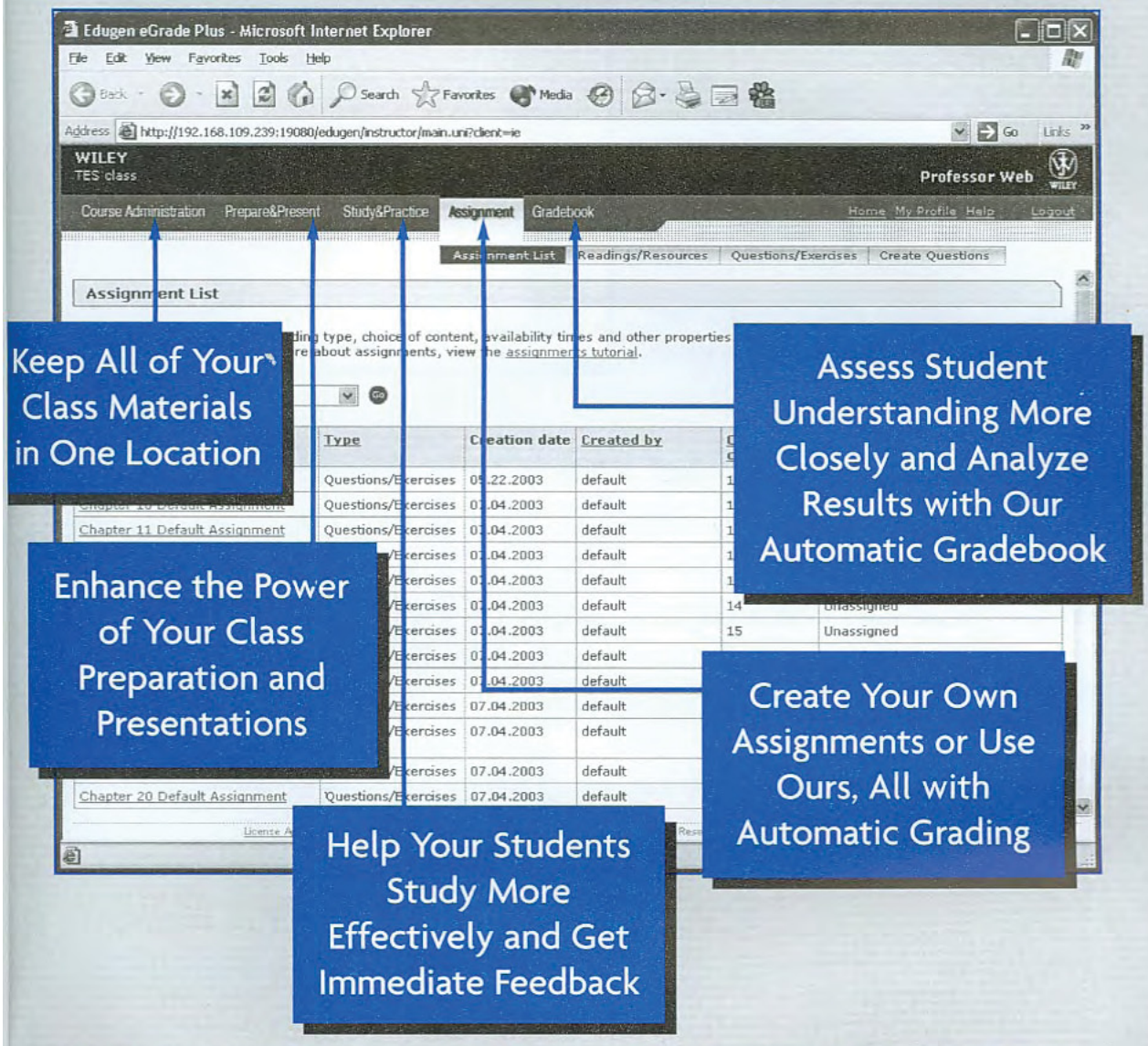
**Greg Gagne** is chair of the Division of Computer Science and Mathematics at Westminster College in Salt Lake City where he has been teaching since 1990. In addition to teaching operating systems, he also teaches computer networks, distributed systems, object-oriented programming, and data structures. He also provides workshops to computer science educators and industry professionals. Professor Gagne's current research interests include next-generation operating systems and distributed computing.

**Peter Baer Galvin** is the chief technologist for Corporate Technologies (www.cptech.com). Before that, Peter was the systems manager for Brown University's Computer Science Department. He is also contributing editor for *SysAdmin* magazine. Mr. Galvin has written articles for *Byte* and other magazines, and previously wrote the security column and systems administration column for *ITWorld*. As a consultant and trainer, Peter has given talks and taught tutorials on security and system administration worldwide.

# eGrade Plus

## www.wiley.com/college/silberschatz

## Based on the Activities You Do Every Day




**All the content and tools you need, all in one location, in an easy-to-use browser format. Choose the resources you need, or rely on the arrangement supplied by us.**

Now, many of Wiley's textbooks are available with eGrade Plus, a powerful online tool that provides a completely integrated suite of teaching and learning resources in one easy-to-use website. eGrade Plus integrates Wiley's world-renowned content with media, including a multimedia version of the text. Upon adoption of eGrade Plus, you can begin to customize your course with the resources shown here.

**See for yourself!**

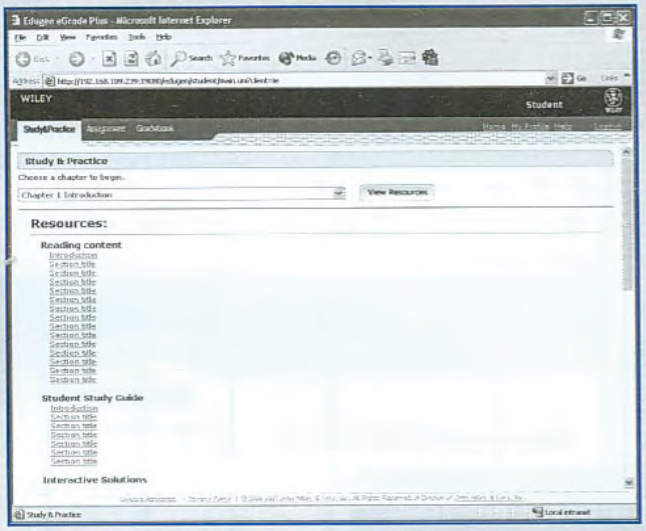
**Go to www.wiley.com/college/egradeplus for an online demonstration of this powerful new software.**

# Students, eGrade Plus Allows You to:

## Study More Effectively

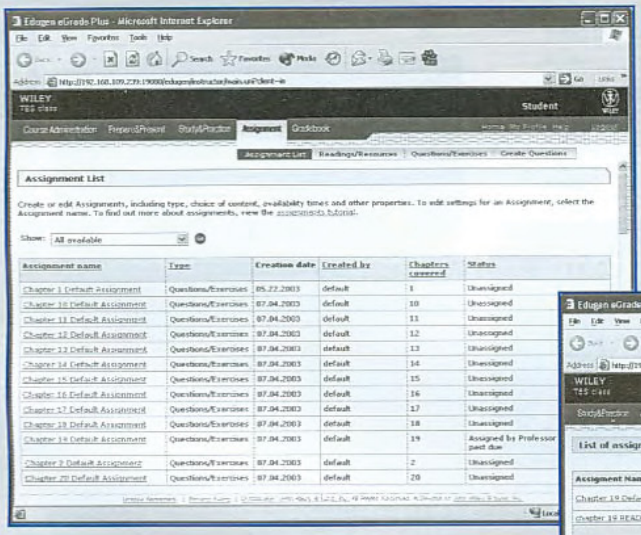
## Get Immediate Feedback When You Practice on Your Own

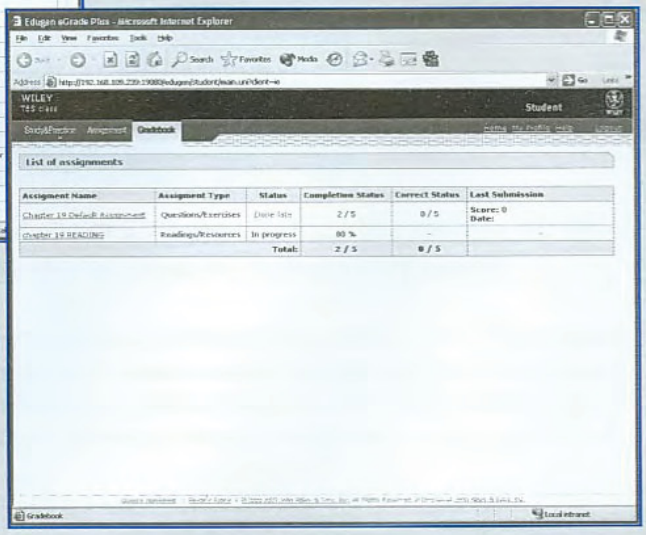
Our website links directly to **electronic book content,** so that you can review the text while you study and complete homework online. Additional resources include **self-assessment quizzing with detailed feedback, source code** and **presentation slides.**




## Complete Assignments / Get Help with Problem Solving

An "**Assignment**" area keeps all your assigned work in one location, making it easy for you to stay on task. In addition, many homework problems contain a **link** to the relevant section of the **electronic book,** providing you with a text explanation to help you conquer problem-solving obstacles as they arise.




## Keep Track of How You're Doing

A **Personal Gradebook** allows you to view your results from past assignments at any time.

# 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

*To my children, Lemor, Sivan, and Aaron*

*Avi Silberschatz*

*To my wife, Carla,*
*and my children, Gwen Owen and Maddie*

*Peter Baer Galvin*

*In memory of Uncle Sonny,*
*Robert Jon Heileman 1933 — 2004*

*Greg Gagne*

# 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 (POSIX1.b) and an extension for a threads library (POSIX1.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 APIsas 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 avaialble only to faculty who use this text. You can find your representative at the "Find a Rep?" web page: http://www.jsw-edcv.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 Huangs, 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
1.2 Computer-System Organization 6
1.3 Computer-System Architecture 12
1.4 Operating-System Structure 15
1.5 Operating-System Operations 17
1.6 Process Management 20
1.7 Memory Management 21
1.8 Storage Management 22
1.9 Protection and Security 26
1.10 Distributed Systems 28
1.11 Special-Purpose Systems 29
1.12 Computing Environments 31
1.13 Summary 34
Exercises 36
Bibliographical Notes 38

### Chapter 2 Operating-System Structures

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

## PART TWO ■ PROCESS MANAGEMENT

### Chapter 3 Processes

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

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

## Chapter 15 Security

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

# PART SIX ■ DISTRIBUTED SYSTEMS

## Chapter 16 Distributed System Structures

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

## Chapter 17 Distributed File Systems

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

## Chapter 18 Distributed Coordination

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

# PART SEVEN ■ SPECIAL-PURPOSE SYSTEMS

## Chapter 19 Real-Time Systems

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

## Chapter 20 Multimedia Systems

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

# PART EIGHT ■ CASE STUDIES

## Chapter 21 The Linux System

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

## Chapter 22 Windows XP

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

## Chapter 23 Influential Operating Systems

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

## Appendix A UNIX BSD (contents online)

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

## Appendix B The Mach System (contents online)

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

## Appendix C Windows 2000 (contents online)

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

## Bibliography 855

## Credits 885

## Index 887

# Index


**2PC protocol,** *see* two-phase commit protocol
**10BaseT Ethernet,** 619
**16-bit Windows environment,** 812
**32-bit Windows environment,** 812–813
**100BaseT Ethernet,** 619

**A**

**aborted transactions,** 222
**absolute code,** 278
**absolute path names,** 390
**abstract data type,** 375
**access:**
  anonymous, 398
  controlled, 402–403
  file, *see* file access
**access control, in Linux,** 778–779
**access-control list (ACL),** 403
**access latency,** 484
**access lists (NFS V4),** 656
**access matrix,** 538–542
  and access control, 545–546
  defined, 538
  implementation of, 542–545
  and revocation of access rights, 546–547
**access rights,** 534, 546–547
**accounting (operating system service),** 41
**accreditation,** 602
**ACL (access-control list),** 403
**active array (Linux),** 752
**Active Directory (Windows XP),** 828
**active list,** 685
**acyclic graph,** 392
**acyclic-graph directories,** 391–394
**adaptive mutex,** 218–219
**additional-reference-bits algorithm,** 336
**additional sense code,** 515
**additional sense-code qualifier,** 515
**address(es):**
  defined, 501
  Internet, 623
  linear, 306
  logical, 279
  physical, 279
  virtual, 279
**address binding,** 278–279
**address resolution protocol (ARP),** 636
**address space:**
  logical vs. physical, 279–280
  virtual, 317, 760–761
**address-space identifiers (ASIDs),** 293–294
**administrative complexity,** 645
**admission control,** 721, 729
**admission-control algorithms,** 704
**advanced encryption standard (AES),** 579
**advanced technology attachment (ATA) buses,** 453
**advisory file-locking mechanisms,** 379
**AES (advanced encryption standard),** 579
**affinity, processor,** 170

**aging,** 163–164, 636
**allocation:**
buddy-system, 354–355
of disk space, 421–429
contiguous allocation, 421–423
indexed allocation, 425–427
linked allocation, 423–425
and performance, 427–429
equal, 341
as problem, 384
proportional, 341
slab, 355–356
**analytic evaluation,** 181
**Andrew file system (AFS),** 653–659
file operations in, 657–658
implementation of, 658–659
shared name space in, 656–657
**anomaly detection,** 595
**anonymous access,** 398
**anonymous memory,** 467
**APCs,** *see* asynchronous procedure calls
**API,** *see* application program interface
**Apple Computers,** 42
**AppleTalk protocol,** 824
**Application Domain,** 69
**application interface (I/O systems),** 505–511
block and character devices, 507–508
blocking and nonblocking I/O, 510–511
clocks and timers, 509–510
network devices, 508–509
**application layer,** 629
**application programs,** 4
disinfection of, 596–597
multistep processing of, 278, 279
processes vs., 21
system utilities, 55–56
**application program interface (API),** 44–46
**application proxy firewalls,** 600
**arbitrated loop (FC-AL),** 455
**architecture(s),** 12–15
clustered systems, 14–15
multiprocessor systems, 12–13
single-processor systems, 12–14
of Windows XP, 787–788
**architecture state,** 171
**archived to tape,** 480
**areal density,** 492
**argument vector,** 749
**armored viruses,** 571
**ARP (address resolution protocol),** 636
**arrays,** 316
**ASIDs,** *see* address-space identifiers
**assignment edge,** 249
**asymmetric clustering,** 15
**asymmetric encryption,** 580
**asymmetric multiprocessing,** 13, 169
**asynchronous devices,** 506, 507
**asynchronous (nonblocking) message passing,** 102
**asynchronous procedure calls (APCs),** 140–141, 790–791
**asynchronous thread cancellation,** 139
**asynchronous writes,** 434
**ATA buses,** 453
**Atlas operating system,** 845–846
**atomicity,** 669–672
**atomic transactions,** 198, 222–230
and checkpoints, 224–225
concurrent, 225–230
and locking protocols, 227–228
and serializability, 225–227
and timestamp-based protocols, 228–230
system model for, 222–223
write-ahead logging of, 223–224
**attacks,** 560. *See also* denial-of-service attacks
man-in-the-middle, 561
replay, 560
zero-day, 595
**attributes,** 815
**authentication:**
breaching of, 560
and encryption, 580–583
in Linux, 777
two-factor, 591
in Windows, 814
**automatic job sequencing,** 841
**automatic variables,** 566
**automatic work-set trimming (Windows XP),** 363
**automount feature,** 645
**autoprobes,** 747
**auxiliary rights (Hydra),** 548

## B

**back door,** 507
**background processes,** 166
**backing store,** 282
**backups,** 436
**bad blocks,** 464–465
**bandwidth:**
    disk, 457
    effective, 484
    sustained, 484
**banker's algorithm,** 259–262
**base file record,** 815
**base register,** 276, 277
**basic file systems,** 412
**batch files,** 379
**batch interface,** 41
**Bayes' theorem,** 596
**Belady's anomaly,** 332
**best-fit strategy,** 287
**biased protocol,** 674
**binary semaphore,** 201
**binding,** 278
**biometrics,** 591–592
**bit(s):**
    mode, 18
    modify (dirty), 329
    reference, 336
    valid-invalid, 295–296
**bit-interleaved parity organization,** 472
**bit-level striping,** 470
**bit vector (bit map),** 429
**black-box transformations,** 579
**blade servers,** 14
**block(s),** 47, 286, 382
    bad, 464–465
    boot, 71, 463–464
    boot control, 414
    defined, 772
    direct, 427
    file-control, 413
    index, 426
    index to, 384
    indirect, 427
    logical, 454
    volume control, 414
**block ciphers,** 579
**block devices,** 506–508, 771–772
**block groups,** 767
**blocking, indefinite,** 163
**blocking I/O,** 510–511
**blocking (synchronous) message passing,** 102
**block-interleaved distributed parity,** 473
**block-interleaved parity organization,** 472–473
**block-level striping,** 470
**block number, relative,** 383–384
**boot block,** 71, 414, 463–464
**boot control block,** 414
**boot disk (system disk),** 72, 464
**booting,** 71–72, 810–811
**boot partition,** 464
**boot sector,** 464
**bootstrap programs,** 463–464, 573
**bootstrap programs (bootstrap loaders),** 6, 7, 71
**boot viruses,** 569
**bottom half interrupt service routines,** 755
**bounded-buffer problem,** 205
**bounded capacity (of queue),** 102
**breach of availability,** 560
**breach of confidentiality,** 560
**breach of integrity,** 560
**broadcasting,** 636, 725
**B+ tree (NTFS),** 816
**buddy heap (Linux),** 757
**buddy system (Linux),** 757
**buddy-system allocation,** 354–355
**buffer,** 772
    circular, 438
    defined, 512
**buffer cache,** 433
**buffering,** 102, 512–514, 729
**buffer-overflow attacks,** 565–568
**bully algorithm,** 684–685
**bus,** 453
    defined, 496
    expansion, 496
    PCI, 496
**bus architecture,** 11
**bus-mastering I/O boards,** 503
**busy waiting,** 202, 499
**bytecode,** 68
**Byzantine generals problem,** 686

## C

**cache:**
buffer, 433
defined, 514
in Linux, 758
as memory buffer, 277
nonvolatile RAM, 470
page, 433
and performance improvement, 433
and remote file access:
and consistency, 649–650
location of cache, 647–648
update policy, 648, 649
slabs in, 355
unified buffer, 433, 434
in Windows XP, 806–808
**cache coherency,** 26
**cache-consistency problem,** 647
**cachefs file system,** 648
**cache management,** 24
**caching,** 24–26, 514
client-side, 827
double, 433
remote service vs., 650–651
write-back, 648
**callbacks,** 657
**Cambridge CAP system,** 549–550
**cancellation, thread,** 139
**cancellation points,** 139
**capability(-ies),** 543, 549
**capability-based protection systems,** 547–550
Cambridge CAP system, 549–550
Hydra, 547–549
**capability lists,** 543
**carrier sense with multiple access (CSMA),** 627–628
**cascading termination,** 95
**CAV (constant angular velocity),** 454
**CD,** *see* collision detection
**central processing unit,** *see under* CPU
**certificate authorities,** 584
**certification,** 602
**challenging (passwords),** 590
**change journal (Windows XP),** 821
**character devices (Linux),** 771–773
**character-stream devices,** 506–508
**checkpoints,** 225
**checksum,** 637
**child processes,** 796
**children,** 90
**CIFS (common internet file system),** 399
**CineBlitz,** 728–730
**cipher-block chaining,** 579
**circuit switching,** 626–627
**circular buffer,** 438
**circular SCAN (C-SCAN) scheduling algorithm,** 460
**circular-wait condition (deadlocks),** 254–256
**claim edge,** 258
**classes (Java),** 553
**class loader,** 68
**CLI (command-line interface),** 41
**C library,** 49
**client(s):**
defined, 642
diskless, 644
in SSL, 586
**client interface,** 642
**client-server model,** 398–399
**client-side caching (CSC),** 827
**client systems,** 31
**clock, logical,** 665
**clock algorithm,** *see* second-chance page-replacement algorithm
**clocks,** 509–510
**C-LOOK scheduling algorithm,** 461
**close() operation,** 376
**clusters,** 463, 634, 815
**clustered page tables,** 300
**clustered systems,** 14–15
**clustering,** 634
asymmetric, 15
in Windows XP, 363
**cluster remapping,** 820
**cluster server,** 655
**CLV (constant linear velocity),** 454
**code:**
absolute, 278
reentrant, 296
**code books,** 591
**collisions (of file names),** 420
**collision detection (CD),** 627–628
**COM,** *see* component object model
**combined scheme index block,** 427
**command interpreter,** 41–42
**command-line interface (CLI),** 41
**commit protocol,** 669

**committed transactions,** 222
**common internet file system (CIFS),** 399
**communication(s):**
direct, 100
in distributed operating systems, 613
indirect, 100
interprocess, *see* interprocess communication
systems programs for, 55
unreliable, 686–687
**communications (operating system service),** 40
**communication links,** 99
**communication processors,** 619
**communications sessions,** 626
**communication system calls,** 54–55
**compaction,** 288, 422
**compiler-based enforcement,** 550–553
**compile time,** 278
**complexity, administrative,** 645
**component object model (COM),** 825–826
**component units,** 642
**compression:**
in multimedia systems, 718–720
in Windows XP, 821
**compression ratio,** 718
**compression units,** 821
**computation migration,** 616
**computation speedup,** 612
**computer environments,** 31–34
client-server computing, 32–33
peer-to-peer computing, 33–34
traditional, 31–32
Web-based computing, 34
**computer programs,** *see* application programs
**computer system(s):**
architecture of:
clustered systems, 14–15
multiprocessor systems, 12–13
single-processor systems, 12–14
distributed systems, 28–29
file-system management in, 22–23
I/O structure in, 10–11
memory management in, 21–22
operating system viewed by, 5
operation of, 6–8
process management in, 20–21
protection in, 26–27
secure, 560
security in, 27
special-purpose systems, 29–31
handheld systems, 30–31
multimedia systems, 30
real-time embedded systems, 29–30
storage in, 8–10
storage management in, 22–26
caching, 24–26
I/O systems, 26
mass-storage management, 23–24
threats to, 571–572
**computing, safe,** 598
**concurrency control,** 672–676
with locking protocols, 672–675
with timestamping, 675–676
**concurrency-control algorithms,** 226
**conditional-wait construct,** 215
**confidentiality, breach of,** 560
**confinement problem,** 541
**conflicting operations,** 226
**conflict phase (of dispatch latency),** 703
**conflict resolution module (Linux),** 747–748
**connectionless messages,** 626
**connectionless (UDP) sockets,** 109
**connection-oriented (TCP) sockets,** 109
**conservative timestamp-ordering scheme,** 676
**consistency,** 649–650
**consistency checking,** 435–436
**consistency semantics,** 401
**constant angular velocity (CAV),** 454
**constant linear velocity (CLV),** 454
**container objects (Windows XP),** 603
**contention,** 627–628
**contention scope,** 172
**context (of process),** 89
**context switches,** 90, 522–523
**contiguous disk space allocation,** 421–423
**contiguous memory allocation,** 285
**continuous-media data,** 716
**control cards,** 49, 842, 843
**control-card interpreter,** 842
**controlled access,** 402–403

**controller(s),** 453, 496–497
defined, 496
direct-memory-access, 503
disk, 453
host, 453
**control programs,** 5
**control register,** 498
**convenience,** 3
**convoy effect,** 159
**cooperating processes,** 96
**cooperative scheduling,** 156
**copy-on-write technique,** 325–327
**copy semantics,** 513
**core memory,** 846
**counting,** 431
**counting-based page replacement algorithm,** 338
**counting semaphore,** 201
**covert channels,** 564
**CPU (central processing unit),** 4, 275–277
**CPU-bound processes,** 88–89
**CPU burst,** 154
**CPU clock,** 276
**CPU-I/O burst cycle,** 154–155
**CPU scheduler,** *see* short-term scheduler
**CPU scheduling,** 17
about, 153–154
algorithms for, 157–169
criteria, 157–158
evaluation of, 181–185
first-come, first-served scheduling of, 158–159
implementation of, 184–185
multilevel feedback-queue scheduling of, 168–169
multilevel queue scheduling of, 166–167
priority scheduling of, 162–164
round-robin scheduling of, 164–166
shortest-job-first scheduling of, 159–162
dispatcher, role of, 157
and I/O-CPU burst cycle, 154–155
models for, 181–185
deterministic modeling, 181–182
and implementation, 184–185
queueing-network analysis, 183
simulations, 183–184
in multimedia systems, 722–723
multiprocessor scheduling, 169–172
approaches to, 169–170
and load balancing, 170–171
and processor affinity, 170
symmetric multithreading, 171–172
preemptive scheduling, 155–156
in real-time systems, 704–710
earliest-deadline-first scheduling, 707
proportional share scheduling, 708
Pthread scheduling, 708–710
rate-monotonic scheduling, 705–707
short-term scheduler, role of, 155
**crackers,** 560
**creation:**
of files, 375
process, 90–95
**critical sections,** 193
**critical-section problem,** 193–195
Peterson's solution to, 195–197
and semaphores, 200–204
deadlocks, 204
implementation, 202–204
starvation, 204
usage, 201
and synchronization hardware, 197–200
**cross-link trust,** 828
**cryptography,** 576–587
and encryption, 577–584
implementation of, 584–585
SSL example of, 585–587
**CSC (client-side caching),** 827
**C-SCAN scheduling algorithm,** 460
**CSMA,** *see* carrier sense with multiple access
**CTSS operating system,** 849
**current directory,** 390
**current-file-position pointer,** 375
**cycles:**
in CineBlitz, 728
CPU-I/O burst, 154–155
**cycle stealing,** 504
**cylinder groups,** 767

## D

**d (page offset),** 289
**daemon process,** 536
**daisy chain,** 496
**data:**
multimedia, 30
recovery of, 435–437
thread-specific, 142
**database systems,** 222
**data capability,** 549
**data-encryption standard (DES),** 579
**data files,** 374
**data fork,** 381
**datagrams,** 626
**data-in register,** 498
**data-link layer,** 629
**data loss, mean time to,** 469
**data migration,** 615–616
**data-out register,** 498
**data section (of process),** 82
**data striping,** 470
**DCOM,** 826
**DDOS attacks,** 560
**deadline I/O scheduler,** 772
**deadlock(s),** 204, 676–683
avoidance of, 252, 256–262
with banker's algorithm, 259–262
with resource-allocation-graph algorithm, 258–259
with safe-state algorithm, 256–258
defined, 245
detection of, 262–265, 678–683
algorithm usage, 265
several instances of a resource type, 263–265
single instance of each resource type, 262–263
methods for handling, 252–253
with mutex locks, 247–248
necessary conditions for, 247–249
prevention/avoidance of, 676–678
prevention of, 252–256
and circular-wait condition, 254–256
and hold-and-wait condition, 253–254
and mutual-exclusion condition, 253
and no-preemption condition, 254
recovery from, 266–267
by process termination, 266
by resource preemption, 267
system model for, 245–247
system resource-allocation graphs for describing, 249–251
**deadlock-detection coordinator,** 679
**debuggers,** 47, 48
**dedicated devices,** 506, 507
**default signal handlers,** 140
**deferred procedure calls (DPCs),** 791
**deferred thread cancellation,** 139
**degree of multiprogramming,** 88
**delay,** 721
**delay-write policy,** 648
**delegation (NFS V4),** 653
**deletion, file,** 375
**demand paging,** 319–325
basic mechanism, 320–322
defined, 319
with inverted page tables, 359–360
and I/O interlock, 361–362
and page size, 357–358
and performance, 323–325
and prepaging, 357
and program structure, 360–361
pure, 322
and restarting instructions, 322–323
and TLB reach, 358–359
**demand-zero memory,** 760
**demilitarized zone (DMZ),** 599
**denial-of-service (DOS) attacks,** 560, 575–576
**density, areal,** 492
**dentry objects,** 419, 765
**DES (data-encryption standard),** 579
**design of operating systems:**
distributed operating systems, 633–636
goals, 56
Linux, 742–744
mechanisms and policies, 56–57
Windows XP, 785–787
**desktop,** 42
**deterministic modeling,** 181–182

**development kernels (Linux),** 739
**device controllers,** 6, 518. *See also* I/O systems
**device directory,** 386. *See also* directories
**device drivers,** 10, 11, 412, 496, 518, 842
**device-management system calls,** 53
**device queues,** 86–87
**device reservation,** 514–515
**DFS,** *see* distributed file system
**digital certificates,** 583–584
**digital signatures,** 582
**digital-signature algorithm,** 582
**dining-philosophers problem,** 207–209, 212–214
**direct access (files),** 383–384
**direct blocks,** 427
**direct communication,** 100
**direct I/O,** 508
**direct memory access (DMA),** 11, 503–504
**direct-memory-access (DMA) controller,** 503
**directories,** 385–387
  acyclic-graph, 391–394
  general graph, 394–395
  implementation of, 419–420
  recovery of, 435–437
  single-level, 387
  tree-structured, 389–391
  two-level, 388–389
**directory objects (Windows XP),** 794
**direct virtual memory access (DVMA),** 504
**dirty bits (modify bits),** 329
**disinfection, program,** 596–597
**disk(s),** 451–453. *See also* mass-storage structure
  allocation of space on, 421–429
    contiguous allocation, 421–423
    indexed allocation, 425–427
    linked allocation, 423–425
    and performance, 427–429
  bad blocks, 464–46
  boot, 72, 464
  boot block, 463–464
  efficient use of, 431
  electronic, 10
  floppy, 452–453
  formatting, 462–463
  free-space management for, 429–431
  host-attached, 455
  low-level formatted, 454
  magnetic, 9
  magneto-optic, 479
  network-attached, 455–456
  performance improvement for, 432–435
  phase-change, 479
  raw, 339
  read-only, 480
  read-write, 479
  removable, 478–480
  scheduling algorithms, 456–462
    C-SCAN, 460
    FCFS, 457–458
    LOOK, 460–461
    SCAN, 459–460
    selecting, 461–462
    SSTF, 458–459
  solid-state, 24
  storage-area network, 456
  structure of, 454
  system, 464
  WORM, 479
**disk arm,** 452
**disk controller,** 453
**diskless clients,** 644
**disk mirroring,** 820
**disk scheduling:**
  CineBlitz, 728
  in multimedia systems, 723–724
**disk striping,** 818
**dispatched process,** 87
**dispatcher,** 157
**dispatcher objects,** 220
  Windows XP, 790
  in Windows XP, 793
**dispatch latency,** 157, 703
**distributed coordination:**
  and atomicity, 669–672
  and concurrency control, 672–676
  and deadlocks, 676–683
    detection, 678–683
    prevention/avoidance, 676–678
  election algorithms for, 683–686
  and event ordering, 663–666
  and mutual exclusion, 666–668
  reaching algorithms for, 686–688
**distributed denial-of-service (DDOS) attacks,** 560

**distributed file system (DFS),** 398
stateless, 401
Windows XP, 827
**distributed file systems (DFSs),** 641–642
AFS example of, 653–659
file operations, 657–658
implementation, 658–659
shared name space, 656–657
defined, 641
naming in, 643–646
remote file access in, 646–651
basic scheme for, 647
and cache location, 647–648
and cache-update policy, 648, 649
and caching vs. remote service, 650–651
and consistency, 649–650
replication of files in, 652–653
stateful vs. stateless service in, 651–652
**distributed information systems (distributed naming services),** 399
**distributed lock manager (DLM),** 15
**distributed naming services,** *see* distributed information systems
**distributed operating systems,** 615–617
**distributed-processing mechanisms,** 824–826
**distributed systems,** 28–29
benefits of, 611–613
defined, 611
distributed operating systems as, 615–617
network operating systems as, 613–615
**DLLs,** *see* dynamic link libraries
**DLM (distributed lock manager),** 15
**DMA,** *see* direct memory access
**DMA controller,** *see* direct-memory-access controller
**DMZ (demilitarized zone),** 599
**domains,** 400, 827–828
**domain-name system (DNS),** 399, 623
**domain switching,** 535
**domain trees,** 827
**DOS attacks,** *see* denial-of-service attacks
**double buffering,** 513, 729
**double caching,** 433
**double indirect blocks,** 427
**downsizing,** 613
**down time,** 422
**DPCs (deferred procedure calls),** 791
**DRAM,** *see* dynamic random-access memory
**driver end (STREAM),** 520
**driver registration module (Linux),** 746–747
**dual-booted systems,** 417
**dumpster diving,** 562
**duplex set,** 820
**DVMA (direct virtual memory access),** 504
**dynamic linking,** 764
**dynamic link libraries (DLLs),** 281–282, 787
**dynamic loading,** 280–281
**dynamic priority,** 722
**dynamic protection,** 534
**dynamic random-access memory (DRAM),** 8
**dynamic routing,** 625
**dynamic storage-allocation problem,** 286, 422

## E

**earliest-deadline-first (EDF) scheduling,** 707, 723
**ease of use,** 4, 784
**ECC,** *see* error-correcting code
**EDF scheduling,** *see* earliest-deadline-first scheduling
**effective access time,** 323
**effective bandwidth,** 484
**effective memory-access time,** 294
**effective UID,** 27
**efficiency,** 3, 431–432
**EIDE buses,** 453
**election,** 628
**election algorithms,** 683–686
**electronic disk,** 10
**elevator algorithm,** *see* SCAN scheduling algorithm
**embedded systems,** 696
**encapsulation (Java),** 555
**encoded files,** 718
**encrypted passwords,** 589–590
**encrypted viruses,** 570

**encryption,** 577–584
asymmetric, 580
authentication, 580–583
key distribution, 583–584
symmetric, 579–580
Windows XP, 821
**enhanced integrated drive electronics (EIDE) buses,** 453
**entry section,** 193
**entry set,** 218
**environmental subsystems,** 786–787
**environment vector,** 749
**EPROM (erasable programmable read-only memory),** 71
**equal allocation,** 341
**erasable programmable read-only memory (EPROM),** 71
**error(s),** 515
hard, 465
soft, 463
**error conditions,** 316
**error-correcting code (ECC),** 462, 471
**error detection,** 40
**escalate privileges,** 27
**escape (operating systems),** 507
**events,** 220
**event latency,** 702
**event objects (Windows XP),** 790
**event ordering,** 663–666
**exceptions (with interrupts),** 501
**exclusive lock mode,** 672
**exclusive locks,** 378
**exec() system call,** 138
**executable files,** 82, 374
**execution of user programs,** 762–764
**execution time,** 278
**exit section,** 193
**expansion bus,** 496
**expired array (Linux),** 752
**expired tasks (Linux),** 752
**exponential average,** 161
**export list,** 441–442
**ext2fs,** *see* second extended file system
**extended file system,** 413, 766
**extent (contiguous space),** 423
**extents,** 815
**external data representation (XDR),** 112
**external fragmentation,** 287–288, 422

**F**

**failure:**
detection of, 631–633
mean time to, 468
recovery from, 633
during writing of block, 477–478
**failure handling (2PC protocol),** 670–672
**failure modes (directories),** 400–401
**fair share (Solaris),** 176
**false negatives,** 595
**false positives,** 595
**fast I/O mechanism,** 807
**FAT (file-allocation table),** 425
**fault tolerance,** 13, 634, 818–821
**fault-tolerant systems,** 634
**FC (fiber channel),** 455
**FC-AL (arbitrated loop),** 455
**FCB (file-control block),** 413
**FC buses,** 453
**FCFS scheduling algorithm,** *see* first-come, first-served scheduling algorithm
**fibers,** 832
**fiber channel (FC),** 455
**fiber channel (FC) buses,** 453
**fids (NFS V4),** 656
**FIFO page replacement algorithm,** 331–333
**50-percent rule,** 287
**file(s),** 22, 373–374. *See also* directories
accessing information on, 382–384
direct access, 383–384
sequential access, 382–383
attributes of, 374–375
batch, 379
defined, 374
executable, 82
extensions of, 379–390
internal structure of, 381–382
locking open, 377–379
operations on, 375–377
protecting, 402–407
via file access, 402–406
via passwords/permissions, 406–407
recovery of, 435–437
storage structure for, 385–386

**file access,** 377, 402–406
**file-allocation table (FAT),** 425
**file-control block (FCB),** 413
**file descriptor,** 415
**file handle,** 415
**FileLock (Java),** 377
**file management,** 55
**file-management system calls,** 53
**file mapping,** 350
**file migration,** 643
**file modification,** 55
**file objects,** 419, 765
**file-organization module,** 413
**file pointers,** 377
**file reference,** 815
**file replication (distributed file systems),** 652–654
**file-server systems,** 31
**file session,** 401
**file sharing,** 397–402
  and consistency semantics, 401–402
  with multiple users, 397–398
  with networks, 398–401
    and client-server model, 398–399
    and distributed information systems, 399–400
    and failure modes, 400–401
**file systems,** 373, 411–413
  basic, 412
  creation of, 386
  design problems with, 412
  distributed, 398, *see* distributed file systems
  extended, 412
  implementation of, 413–419
    mounting, 417
    partitions, 416–417
    virtual systems, 417–419
  levels of, 412
  Linux, 764–770
  log-based transaction-oriented, 437–438
  logical, 412
  mounting of, 395–397
  network, 438–444
  remote, 398
  WAFL, 444–446
**File System Hierarchy Standard document,** 740
**file-system management,** 22–23
**file-system manipulation (operating system service),** 40
**file transfer,** 614–615
**file transfer protocol (FTP),** 614–615
**file viruses,** 569
**filter drivers,** 806
**firewalls,** 31, 599–600
**firewall chains,** 776
**firewall management,** 776
**FireWire,** 454
**firmware,** 6, 71
**first-come, first-served (FCFS) scheduling algorithm,** 158–159, 457–458
**first-fit strategy,** 287
**fixed-partition scheme,** 286
**fixed priority (Solaris),** 176
**fixed routing,** 625
**floppy disks,** 452–453
**flow control,** 521
**flushing,** 294
**folders,** 42
**footprint,** 697
**foreground processes,** 166
**forests,** 827–828
**fork() and exec() process model (Linux),** 748–750
**fork() system call,** 138
**formatting,** 462–463
**forwarding,** 465
**forward-mapped page tables,** 298
**fragments, packet,** 776
**fragmentation,** 287–288
  external, 287–288, 422
  internal, 287, 382
**frame(s),** 289, 626, 716
  stack, 566–567
  victim, 329
**frame allocation,** 340–343
  equal allocation, 341
  global vs. local, 342–343
  proportional allocation, 341–342
**frame-allocation algorithm,** 330
**frame pointers,** 567
**free-behind technique,** 435
**free objects,** 356, 758

**free-space list,** 429
**free-space management (disks),** 429–431
    bit vector, 429–430
    counting, 431
    grouping, 431
    linked list, 430–431
**front-end processors,** 523
FTP, *see* file transfer protocol
**ftp,** 398
**full backup,** 436
**fully distributed deadlock-detection algorithm,** 681–683

**G**

**Gantt chart,** 159
**garbage collection,** 68, 395
**gateways,** 626
**GB (gigabyte),** 6
**gcc (GNU C compiler),** 740
**GDT (global descriptor table),** 306
**general graph directories,** 394–395
**gigabyte (GB),** 6
**global descriptor table (GDT),** 306
**global ordering,** 665
**global replacement,** 342
**GNU C compiler (gcc),** 740
**GNU Portable Threads,** 130
**graceful degradation,** 13
**graphs, acyclic,** 392
**graphical user interfaces (GUIs),** 41–43
**grappling hook,** 573
**Green threads,** 130
**group identifiers,** 27
**grouping,** 431
**group policies,** 828
**group rights (Linux),** 778
**guest operating systems,** 67
GUIs, *see* graphical user interfaces

**H**

HAL, *see* hardware-abstraction layer
**handheld computers,** 5
**handheld systems,** 30–31
**handles,** 793, 796
**handling (of signals),** 123
**handshaking,** 498–499, 518
**hands-on computer systems,** *see* interactive computer systems
**happened-before relation,** 664–666
**hard affinity,** 170
**hard-coding techniques,** 100
**hard errors,** 465
**hard links,** 394
**hard real-time systems,** 696, 722
**hardware,** 4
    I/O systems, 496–505
        direct memory access, 503–504
        interrupts, 499–503
        polling, 498–499
    for storing page tables, 292–294
    synchronization, 197–200
**hardware-abstraction layer (HAL),** 787, 788
**hardware objects,** 533
**hashed page tables,** 300
**hash functions,** 582
**hash tables,** 420
**hash value (message digest),** 582
**heaps,** 82, 835–836
**heavyweight processes,** 127
**hierarchical paging,** 297–300
**hierarchical storage management (HSM),** 483
**high availability,** 14
**high performance,** 786
**hijacking, session,** 561
**hit ratio,** 294, 358
**hive,** 810
**hold-and-wait condition (deadlocks),** 253–254
**holes,** 286
**holographic storage,** 480
**homogeneity,** 169
**host adapter,** 496
**host-attached storage,** 455
**host controller,** 453
**hot spare disks,** 475
**hot-standby mode,** 15
**HSM (hierarchical storage management),** 483
**human security,** 562
**Hydra,** 547–549
**hyperspace,** 797
**hyperthreading technology,** 171

I

**IBM OS/360,** 850–851
**identifiers:**
file, 374
group, 27
user, 27
**idle threads,** 177
**IDSs,** *see* intrusion-detection systems
**IKE protocol,** 585
**ILM (information life-cycle management),** 483
**immutable shared files,** 402
**implementation:**
of CPU scheduling algorithms, 184–185
of operating systems, 57–58
of real-time operating systems, 700–704
and minimizing latency, 702–704
and preemptive kernels, 701
and priority-based scheduling, 700–701
of transparent naming techniques, 645–646
of virtual machines, 65–66
**incremental backup,** 436
**indefinite blocking (starvation),** 163, 204
**independence, location,** 643
**independent disks,** 469
**independent processes,** 96
**index,** 384
**index block,** 426
**indexed disk space allocation,** 425–427
**index root,** 816
**indirect blocks,** 427
**indirect communication,** 100
**information life-cycle management (ILM),** 483
**information-maintenance system calls,** 53–54
**inode objects,** 419, 765
**input/output,** *see under* I/O
**input queue,** 278
**InServ storage array,** 476
**instance handles,** 831
**instruction-execution cycle,** 275–276
**instruction-execution unit,** 811
**instruction register,** 8
**integrity, breach of,** 560
**intellimirror,** 828
**Intel Pentium processor,** 305–308
**interactive (hands-on) computer systems,** 16
**interface(s):**
batch, 41
client, 642
defined, 505
intermachine, 642
Windows XP networking, 822
**interlock, I/O,** 361–362
**intermachine interface,** 642
**internal fragmentation,** 287, 382
**international use,** 787
**Internet address,** 623
**Internet Protocol (IP),** 584–585
**interprocess communication (IPC),** 96–102
in client-server systems, 108–115
remote method invocation, 114–115
remote procedure calls, 111–113
sockets, 108–111
in Linux, 739, 773–774
Mach example of, 105–106
in message-passing systems, 99–102
POSIX shared-memory example of, 103–104
in shared-memory systems, 97–99
Windows XP example of, 106–108
**interrupt(s),** 7, 499–503
defined, 499
in Linux, 754–755
**interrupt chaining,** 501
**interrupt-controller hardware,** 501
**interrupt-dispatch table (Windows XP),** 792
**interrupt-driven data transfer,** 353
**interrupt-driven operating systems,** 17–18
**interrupt latency,** 702–703
**interrupt priority levels,** 501
**interrupt-request line,** 499
**interrupt vector,** 8, 284, 501
**intruders,** 560
**intrusion detection,** 594–596
**intrusion-detection systems (IDSs),** 594–595
**intrusion-prevention systems (IPSs),** 595

**inverted page tables,** 301–302, 359–360
**I/O (input/output),** 4, 10–11
    memory-mapped, 353
    overlapped, 843–845
    programmed, 353
**I/O-bound processes,** 88–89
**I/O burst,** 154
**I/O channel,** 523, 524
**I/O interlock,** 361–362
**I/O manager,** 805–806
**I/O operations (operating system service),** 40
**I/O ports,** 353
**I/O request packet (IRP),** 805
**I/O subsystem(s),** 26
    kernels in, 6, 511–518
    procedures supervised by, 517–518
**I/O system(s),** 495–496
    application interface, 505–511
        block and character devices, 507–508
        blocking and nonblocking I/O, 510–511
        clocks and timers, 509–510
        network devices, 508–509
    hardware, 496–505
        direct memory access, 503–504
        interrupts, 499–503
        polling, 498–499
    kernels, 511–518
        buffering, 512–514
        caching, 514
        data structures, 516–517
        error handling, 515
        I/O scheduling, 511–512
        and I/O subsystems, 517–518
        protection, 515–516
        spooling and device reservation, 514–515
    Linux, 770–773
        block devices, 771–772
        character devices, 772–773
    STREAMS mechanism, 520–522
    and system performance, 522–525
    transformation of requests to hardware operations, 518–520
**IP,** *see* Internet Protocol
**IPC,** *see* interprocess communication
**IPSec,** 585
**IPSs (intrusion-prevention systems),** 595
**IRP (I/O request packet),** 805
**ISCSI,** 456
**ISO protocol stack,** 630
**ISO Reference Model,** 585

**J**

**Java:**
    file locking in, 377–378
    language-based protection in, 553–555
    monitors in, 218
**Java threads,** 134–138
**Java Virtual Machine (JVM),** 68
**JIT compiler,** 68
**jitter,** 721
**jobs, processes vs.,** 82
**job objects,** 803
**job pool,** 17
**job queues,** 85
**job scheduler,** 88
**job scheduling,** 17
**journaling,** 768–769
**journaling file systems,** *see* log-based transaction-oriented file systems
**just-in-time (JIT) compiler,** 68
**JVM (Java Virtual Machine),** 68

**K**

**KB (kilobyte),** 6
**Kerberos,** 814
**kernel(s),** 6, 511–518
    buffering, 512–514
    caching, 514
    data structures, 516–517
    error handling, 515
    I/O scheduling, 511–512
    and I/O subsystems, 517–518
    Linux, 743, 744
    multimedia systems, 720–722
    nonpreemptive, 194–195
    preemptive, 194–195, 701
    protection, 515–516
    real-time, 698–700
    spooling and device reservation, 514–515
    task synchronization (in Linux), 753–755
    Windows XP, 788–793, 829

**kernel extensions,** 63
**kernel memory allocation,** 353–356
**kernel mode,** 18, 743
**kernel modules,** 745–748
    conflict resolution, 747–748
    driver registration, 746–747
    management of, 745–746
**kernel threads,** 129
**Kerr effect,** 479
**keys,** 544, 547, 577
    private, 580
    public, 580
**key distribution,** 583–584
**key ring,** 583
**keystreams,** 580
**keystroke logger,** 571
**kilobyte (KB),** 6

L

**language-based protection systems,** 550–555
    compiler-based enforcement, 550–553
    Java, 553–555
**LANs,** *see* local-area networks
**latency, in real-time systems,** 702–704
**layers (of network protocols),** 584
**layered approach (operating system structure),** 59–61
**lazy swapper,** 319
**LCNs (logical cluster numbers),** 815
**LDAP,** *see* lightweight directory-access protocol
**LDT (local descriptor table),** 306
**least-frequently used (LFU) page-replacement algorithm,** 338
**least privilege, principle of,** 532–533
**least-recently-used (LRU) page-replacement algorithm,** 334–336
**levels,** 719
**LFU page-replacement algorithm,** 338
**libraries:**
    Linux system, 743, 744
    shared, 281–282, 318
**licenses, software,** 235
**lightweight directory-access protocol (LDAP),** 400, 828
**limit register,** 276, 277
**linear addresses,** 306
**linear lists (files),** 420
**line discipline,** 772
**link(s):**
    communication, 99
    defined, 392
    hard, 394
    resolving, 392
    symbolic, 794
**linked disk space allocation,** 423–425
**linked lists,** 430–431
**linked scheme index block,** 426–427
**linking, dynamic vs. static,** 281–282, 764
**Linux,** 737–780
    adding system call to Linux kernel (project), 74–78
    design principles for, 742–744
    file systems, 764–770
        ext2fs, 766–768
        journaling, 768–769
        process, 769–770
        virtual, 765–766
    history of, 737–742
        distributions, 740–741
        first kernel, 738–740
        licensing, 741–742
        system description, 740
    interprocess communication, 773–774
    I/O system, 770–773
        block devices, 771–772
        character devices, 772–773
    kernel modules, 745–748
    memory management, 756–764
        execution and loading of user programs, 762–764
        physical memory, 756–759
        virtual memory, 759–762
    network structure, 774–777
    on Pentium systems, 307–309
    process management, 748–757
        fork() and exec() process model, 748–750
        processes and threads, 750–751
    process representation in, 86
    real-time, 711
    scheduling, 751–756
        kernel synchronization, 753–755

**Linux** (*continued*)
process, 751–753
symmetric multiprocessing, 755–756
scheduling example, 179–181
security model, 777–779
access control, 778–779
authentication, 777
swap-space management in, 468
synchronization in, 221
threads example, 144–146
**Linux distributions,** 738, 740–741
**Linux kernel,** 738–740
**Linux system, components of,** 738, 743–744
**lists,** 316
**Little's formula,** 183
**live streaming,** 717
**load balancers,** 34
**load balancing,** 170–171
**loader,** 842
**loading:**
dynamic, 280–281
in Linux, 762–764
**load sharing,** 169, 612
**load time,** 278
**local-area networks (LANs),** 14, 28, 618–619
**local descriptor table (LDT),** 306
**locality model,** 344
**locality of reference,** 322
**local name space,** 655
**local (nonremote) objects,** 115
**local playback,** 716
**local procedure calls (LPCs),** 786, 804–805
**local replacement,** 342
**local replacement algorithm (priority replacement algorithm),** 344
**location, file,** 374
**location independence,** 643
**location-independent file identifiers,** 646
**location transparency,** 643
**lock(s),** 197, 544
advisory, 379
exclusive, 378
in Java API, 377–378
mandatory, 379
mutex, 201, 251–252
reader-writer, 207
shared, 378
**locking protocols,** 227–228, 672–675
**lock-key scheme,** 544
**lock() operation,** 377
**log-based transaction-oriented file systems,** 437–438
**log-file service,** 817
**logging, write-ahead,** 223–224
**logging area,** 817
**logical address,** 279
**logical address space,** 279–280
**logical blocks,** 454
**logical clock,** 665
**logical cluster numbers (LCNs),** 815
**logical file system,** 413
**logical formatting,** 463
**logical memory,** 17, 317. *See also* virtual memory
**logical records,** 383
**logical units,** 455
**login, network,** 399
**long-term scheduler (job scheduler),** 88
**LOOK scheduling algorithm,** 460–461
**loopback,** 111
**lossless compression,** 718
**lossy compression,** 718–719
**low-level formatted disks,** 454
**low-level formatting (disks),** 462–463
**LPCs,** *see* local procedure calls
**LRU-approximation page replacement algorithm,** 336–338

## M

**MAC (message-authentication code),** 582
**MAC (medium access control) address,** 636
**Mach operating system,** 61, 105–106, 851–853
**Macintosh operating system,** 381–382
**macro viruses,** 569
**magic number (files),** 381
**magnetic disk(s),** 9, 451–453. *See also* disk(s)
**magnetic tapes,** 453–454, 480
**magneto-optic disks,** 479
**mailboxes,** 100
**mailbox sets,** 106
**mailslots,** 824
**mainframes,** 5

**main memory,** 8–9
and address binding, 278–279
contiguous allocation of, 284–285
and fragmentation, 287–288
mapping, 285
methods, 286–287
protection, 285
and dynamic linking, 281–282
and dynamic loading, 280–281
and hardware, 276–278
Intel Pentium example:
with Linux, 307–309
paging, 306–308
segmentation, 305–307
and logical vs. physical address space, 279–280
paging for management of, 288–302
basic method, 289–292
hardware, 292–295
hashed page tables, 300
hierarchical paging, 297–300
Intel Pentium example, 306–308
inverted page tables, 301–302
protection, 295–296
and shared pages, 296–297
segmentation for management of, 302–305
basic method, 302–304
hardware, 304–305
Intel Pentium example, 305–307
and swapping, 282–284
**majority protocol,** 673–674
**MANs (metropolitan-area networks),** 28
**mandatory file-locking mechanisms,** 379
**man-in-the-middle attack,** 561
**many-to-many multithreading model,** 130–131
**many-to-one multithreading model,** 129–130
**marshalling,** 825
**maskable interrupts,** 501
**masquerading,** 560
**mass-storage management,** 23–24
**mass-storage structure,** 451–454
disk attachment:
host-attached, 455
network-attached, 455–456
storage-area network, 456
disk management:
bad blocks, 464–46
boot block, 463–464
formatting of disks, 462–463
disk scheduling algorithms, 456–462
C-SCAN, 460
FCFS, 457–458
LOOK, 460–461
SCAN, 459–460
selecting, 461–462
SSTF, 458–459
disk structure, 454
extensions, 476
magnetic disks, 451–453
magnetic tapes, 453–454
RAID structure, 468–477
performance improvement, 470
problems with, 477
RAID levels, 470–476
reliability improvement, 468–470
stable-storage implementation, 477–478
swap-space management, 466–468
tertiary-storage, 478–488
future technology for, 480
magnetic tapes, 480
and operating system support, 480–483
performance issues with, 484–488
removable disks, 478–480
**master book record (MBR),** 464
**master file directory (MFD),** 388
**master file table,** 414
**master key,** 547
**master secret (SSL),** 586
**matchmakers,** 112
**matrix product,** 149
**MB (megabyte),** 6
**MBR (master book record),** 464
**MCP operating system,** 853
**mean time to data loss,** 469
**mean time to failure,** 468
**mean time to repair,** 469
**mechanisms,** 56–57
**media players,** 727
**medium access control (MAC) address,** 636

**medium-term scheduler,** 89
**megabyte (MB),** 6
**memory:**
anonymous, 467
core, 846
direct memory access, 11
direct virtual memory access, 504
logical, 17, 317
main, *see* main memory
over-allocation of, 327
physical, 17
secondary, 322
semiconductor, 10
shared, 96, 318
unified virtual memory, 433
virtual, *see* virtual memory
**memory-address register,** 279
**memory allocation,** 286–287
**memory management,** 21–22
in Linux, 756–764
execution and loading of user programs, 762–764
physical memory, 756–759
virtual memory, 759–762
in Windows XP, 834–836
heaps, 835–836
memory-mapping files, 835
thread-local storage, 836
virtual memory, 834–835
**memory-management unit (MMU),** 279–280, 799
**memory-mapped files,** 798
**memory-mapped I/O,** 353, 497
**memory mapping,** 285, 348–353
basic mechanism, 348–350
defined, 348
I/O, memory-mapped, 353
in Linux, 763–764
in Win32 API, 350–353
**memory-mapping files,** 835
**memory protection,** 285
**memory-resident pages,** 320
**memory-style error-correcting organization,** 471
**MEMS (micro-electronic mechanical systems),** 480
**messages:**
connectionless, 626
in distributed operating systems, 613
**message-authentication code (MAC),** 582
**message digest (hash value),** 582
**message modification,** 560
**message passing,** 96
**message-passing model,** 54, 99–102
**message queue,** 848
**message switching,** 627
**metadata,** 400, 816
**metafiles,** 727
**methods (Java),** 553
**metropolitan-area networks (MANs),** 28
**MFD (master file directory),** 388
**MFU page-replacement algorithm,** 338
**micro-electronic mechanical systems (MEMS),** 480
**microkernels,** 61–64
**Microsoft Interface Definition Language,** 825
**Microsoft Windows,** *see under* Windows
**migration:**
computation, 616
data, 615–616
file, 643
process, 617
**minicomputers,** 5
**minidisks,** 386
**miniport driver,** 806
**mirroring,** 469
**mirror set,** 820
**MMU,** *see* memory-management unit
**mobility, user,** 440
**mode bit,** 18
**modify bits (dirty bits),** 329
**modules,** 62–63, 520
**monitors,** 209–217
dining-philosophers solution using, 212–214
implementation of, using semaphores, 214–215
resumption of processes within, 215–217
usage of, 210–212
**monitor calls,** *see* system calls
**monoculture,** 571
**monotonic,** 665
**Morris, Robert,** 572–574
**most-frequently used (MFU) page-replacement algorithm,** 338
**mounting,** 417
**mount points,** 395, 821
**mount protocol,** 440–441

**mount table,** 417, 518
**MPEG files,** 719
**MS-DOS,** 811–812
**multicasting,** 725
**MULTICS operating system,** 536–538, 849–850
**multilevel feedback-queue scheduling algorithm,** 168–169
**multilevel index,** 427
**multilevel queue scheduling algorithm,** 166–167
**multimedia,** 715–716
  operating system issues with, 718
  as term, 715–716
**multimedia data,** 30, 716–717
**multimedia systems,** 30, 715
  characteristics of, 717–718
  CineBlitz example, 728–730
  compression in, 718–720
  CPU scheduling in, 722–723
  disk scheduling in, 723–724
  kernels in, 720–722
  network management in, 725–728
**multinational use,** 787
**multipartite viruses,** 571
**multiple-coordinator approach (concurrency control),** 673
**multiple-partition method,** 286
**multiple universal-naming-convention provider (MUP),** 826
**multiprocessing:**
  asymmetric, 169
  symmetric, 169, 171–172
**multiprocessor scheduling,** 169–172
  approaches to, 169–170
  examples of:
    Linux, 179–181
    Solaris, 173, 175–177
    Windows XP, 178–179
  and load balancing, 170–171
  and processor affinity, 170
  symmetric multithreading, 171–172
**multiprocessor systems (parallel systems, tightly coupled systems),** 12–13
**multiprogramming,** 15–17, 88
**multitasking,** *see* time sharing
**multithreading:**
  benefits of, 127–129
  cancellation, thread, 139
  and **exec()** system call, 138
  and **fork()** system call, 138
  models of, 129–131
  pools, thread, 141–142
  and scheduler activations, 142–143
  and signal handling, 139–141
  symmetric, 171–172
  and thread-specific data, 142
**MUP (multiple universal-naming-convention provider),** 826
**mutex:**
  adaptive, 218–219
  in Windows XP, 790
**mutex locks,** 201, 247–248
**mutual exclusion,** 247, 666–668
  centralized approach to, 666
  fully-distributed approach to, 666–668
  token-passing approach to, 668
**mutual-exclusion condition (deadlocks),** 253

## N

**names:**
  resolution of, 623, 828–829
  in Windows XP, 793–794
**named pipes,** 824
**naming,** 100–101, 399–400
  defined, 643
  domain name system, 399
  of files, 374
  lightweight diretory-access protocol, 400
  and network communication, 622–625
**national-language-support (NLS) API,** 787
**NDIS (network device interface specification),** 822
**near-line storage,** 480
**negotiation,** 721
**NetBEUI (NetBIOSextended user interface),** 823
**NetBIOS (network basic input/output system),** 823, 824
**NetBIOSextended user interface (NetBEUI),** 823
**.NET Framework,** 69

**network(s).** *See also* local-area networks (LANs); wide-area networks (WANs)
communication protocols in, 628–631
communication structure of, 622–628
and connection strategies, 626–627
and contention, 627–628
and naming/name resolution, 622–625
and packet strategies, 626
and routing strategies, 625–626
defined, 28
design issues with, 633–636
example, 636–637
in Linux, 774–777
metropolitan-area (MANs), 28
robustness of, 631–633
security in, 562
small-area, 28
threats to, 571–572
topology of, 620–622
types of, 617–618
in Windows XP, 822–829
Active Directory, 828
distributed-processing mechanisms, 824–826
domains, 827–828
interfaces, 822
name resolution, 828–829
protocols, 822–824
redirectors and servers, 826–827
wireless, 31
**network-attached storage,** 455–456
**network basic input/output system,** *see* NetBIOS
**network computers,** 32
**network devices,** 508–509, 771
**network device interface specification (NDIS),** 822
**network file systems (NFS),** 438–444
mount protocol, 440–441
NFS protocol, 441–442
path-name translation, 442–443
remote operations, 443–444
**network information service (NIS),** 399
**network layer,** 629
**network-layer protocol,** 584
**network login,** 399
**network management, in multimedia systems,** 725–728
**network operating systems,** 28, 613–615
**network virtual memory,** 647
**new state,** 83
**NFS,** *see* network file systems
**NFS protocol,** 440–442
**NFS V4,** 653
**nice value (Linux),** 179, 752
**NIS (network information service),** 399
**NLS (national-language-support) API,** 787
**nonblocking I/O,** 510–511
**nonblocking (asynchronous) message passing,** 102
**noncontainer objects (Windows XP),** 603
**nonmaskable interrupt,** 501
**nonpreemptive kernels,** 194–195
**nonpreemptive scheduling,** 156
**non-real-time clients,** 728
**nonremote (local) objects,** 115
**nonrepudiation,** 583
**nonresident attributes,** 815
**nonserial schedule,** 226
**nonsignaled state,** 220
**nonvolatile RAM (NVRAM),** 10
**nonvolatile RAM (NVRAM) cache,** 470
**nonvolatile storage,** 10, 223
**no-preemption condition (deadlocks),** 254
**Novell NetWare protocols,** 823
**NTFS,** 814–816
**NVRAM (nonvolatile RAM),** 10
**NVRAM (nonvolatile RAM) cache,** 470

## O

**objects:**
access lists for, 542–543
in cache, 355
free, 356
hardware vs. software, 533
in Linux, 758
used, 356
in Windows XP, 793–796
**object files,** 374

**object linking and embedding (OLE),** 825–826
**object serialization,** 115
**object table,** 796
**object types,** 419, 795
**off-line compaction of space,** 422
**OLE,** *see* object linking and embedding
**on-demand streaming,** 717
**one-time pad,** 591
**one-time passwords,** 590–591
**one-to-one multithreading model,** 130
**one-way trust,** 828
**on-line compaction of space,** 422
**open-file table,** 376
**open() operation,** 376
**operating system(s),** 1
    defined, 3, 5–6
    design goals for, 56
    early, 839–845
        dedicated computer systems, 839–840
        overlapped I/O, 843–845
        shared computer systems, 841–843
    features of, 3
    functioning of, 3–6
    guest, 67
    implementation of, 57–58
    interrupt-driven, 17–18
    mechanisms for, 56–57
    network, 28
    operations of:
        modes, 18–20
        and timer, 20
    policies for, 56–57
    real-time, 29–30
    as resource allocator, 5
    security in, 562
    services provided by, 39–41
    structure of, 15–17, 58–64
        layered approach, 59–61
        microkernels, 61–64
        modules, 62–63
        simple structure, 58–59
    system's view of, 5
    user interface with, 4–5, 41–43
**optimal page replacement algorithm,** 332–334
**ordering, event,** *see* event ordering
**orphan detection and elimination,** 652
**OS/2 operating system,** 783
**out-of-band key delivery,** 583
**over allocation (of memory),** 327
**overlapped I/O,** 843–845
**overprovisioning,** 720
**owner rights (Linux),** 778

## P

**p (page number),** 289
**packets,** 626, 776
**packet switching,** 627
**packing,** 382
**pages:**
    defined, 289
    shared, 296–297
**page allocator (Linux),** 757
**page-buffering algorithms,** 338–339
**page cache,** 433, 759
**page directory,** 799
**page-directory entries (PDEs),** 799
**page-fault-frequency (PFF),** 347–348
**page-fault rate,** 325
**page-fault traps,** 321
**page frames,** 799
**page-frame database,** 801
**page number (p),** 289
**page offset (d),** 289
**pageout (Solaris),** 363–364
**pageout policy (Linux),** 761
**pager (term),** 319
**page replacement,** 327–339. *See also*
    frame allocation
    and application performance, 339
    basic mechanism, 328–331
    counting-based page replacement, 338
    FIFO page replacement, 331–333
    global vs. local, 342
    LRU-approximation page replacement, 336–338
    LRU page replacement, 334–336
    optimal page replacement, 332–334
    and page-buffering algorithms, 338–339
**page replacement algorithm,** 330
**page size,** 357–358
**page slots,** 468

**page table(s),** 289–292, 322, 799
clustered, 300
forward-mapped, 298
hardware for storing, 292–294
hashed, 300
inverted, 301–302, 359–360
**page-table base register (PTBR),** 293
**page-table length register (PTLR),** 296
**page-table self-map,** 797
**paging,** 288–302
basic method of, 289–292
hardware support for, 292–295
hashed page tables, 300
hierarchical, 297–300
Intel Pentium example, 306–308
inverted, 301–302
in Linux, 761–762
and memory protection, 295–296
priority, 365
and shared pages, 296–297
swapping vs., 466
**paging files (Windows XP),** 797
**paging mechanism (Linux),** 761
**paired passwords,** 590
**PAM (pluggable authentication modules),** 777
**parallel systems,** *see* multiprocessor systems
**parcels,** 114
**parent process,** 90, 795–796
**partially connected networks,** 621–622
**partition(s),** 286, 386, 416–417
boot, 464
raw, 467
root, 417
**partition boot sector,** 414
**partitioning, disk,** 463
**passwords,** 588–591
encrypted, 589–590
one-time, 590–591
vulnerabilities of, 588–589
**path name,** 388–389
**path names:**
absolute, 390
relative, 390
**path-name translation,** 442–443
**PCBs,** *see* process control blocks
**PCI bus,** 496
**PCS (process-contention scope),** 172
**PC systems,** 3
**PDAs,** *see* personal digital assistants
**PDEs (page-directory entries),** 799
**peer-to-peer computing,** 33–34
**penetration test,** 592–593
**performance:**
and allocation of disk space, 427–429
and I/O system, 522–525
with tertiary-storage, 484–488
cost, 485–488
reliability, 485
speed, 484–485
of Windows XP, 786
**performance improvement,** 432–435, 470
**periods,** 720
**periodic processes,** 720
**permissions,** 406
**per-process open-file table,** 414
**persistence of vision,** 716
**personal computer (PC) systems,** 3
**personal digital assistants (PDAs),** 10, 30
**personal firewalls,** 600
**personal identification number (PIN),** 591
**Peterson's solution,** 195–197
**PFF,** *see* page-fault-frequency
**phase-change disks,** 479
**phishing,** 562
**physical address,** 279
**physical address space,** 279–280
**physical formatting,** 462
**physical layer,** 628, 629
**physical memory,** 17, 315–316, 756–759
**physical security,** 562
**PIC (position-independent code),** 764
**pid (process identifier),** 90
**PIN (personal identification number),** 591
**pinning,** 807–808
**PIO,** *see* programmed I/O
**pipe mechanism,** 774
**platter (disks),** 451
**plug-and-play and (PnP) managers,** 809–810
**pluggable authentication modules (PAM),** 777
**PnP managers,** *see* plug-and-play and managers

**point-to-point tunneling protocol (PPTP),** 823
**policy(ies),** 56–57
group, 828
security, 592
**policy algorithm (Linux),** 761
**polling,** 498–499
**polymorphic viruses,** 570
**pools:**
of free pages, 327
thread, 141–142
**pop-up browser windows,** 564
**ports,** 353, 496
**portability,** 787
**portals,** 32
**port driver,** 806
**port scanning,** 575
**position-independent code (PIC),** 764
**positioning time (disks),** 452
**POSIX,** 783, 786
interprocess communication
example, 103–104
in Windows XP, 813–814
**possession (of capability),** 543
**power-of-2 allocator,** 354
**PPTP (point-to-point tunneling protocol),** 823
**P + Q redundancy scheme,** 473
**preemption points,** 701
**preemptive kernels,** 194–195, 701
**preemptive scheduling,** 155–156
**premaster secret (SSL),** 586
**prepaging,** 357
**presentation layer,** 629
**primary thread,** 830
**principle of least privilege,** 532–533
**priority-based scheduling,** 700–701
**priority-inheritance protocol,** 219, 704
**priority inversion,** 219, 704
**priority number,** 216
**priority paging,** 365
**priority replacement algorithm,** 344
**priority scheduling algorithm,** 162–164
**private keys,** 580
**privileged instructions,** 19
**privileged mode,** *see* kernel mode
**process(es),** 17
background, 166
communication between, *see* interprocess communication
components of, 82
context of, 89, 749–750
and context switches, 89–90
cooperating, 96
defined, 81
environment of, 749
faulty, 687–688
foreground, 166
heavyweight, 127
independent, 96
I/O-bound vs. CPU-bound, 88–89
job vs., 82
in Linux, 750–751
multithreaded, *see* multithreading
operations on, 90–95
creation, 90–95
termination, 95
programs vs., 21, 82, 83
scheduling of, 85–90
single-threaded, 127
state of, 83
as term, 81–82
threads performed by, 84–85
in Windows XP, 830
**process-contention scope (PCS),** 172
**process control blocks (PCBs, task control blocks),** 83–84
**process-control system calls,** 47–52
**process file systems (Linux),** 769–770
**process identifier (pid),** 90
**process identity (Linux),** 748–749
**process management,** 20–21
in Linux, 748–757
fork() and exec() process model, 748–750
processes and threads, 750–751
**process manager (Windows XP),** 802–804
**process migration,** 617
**process mix,** 88–89
**process objects (Windows XP),** 790
**processor affinity,** 170
**processor sharing,** 165
**process representation (Linux),** 86
**process scheduler,** 85
**process scheduling:**
in Linux, 751–753
thread scheduling vs., 153

**process synchronization:**
about, 191–193
and atomic transactions, 222–230
checkpoints, 224–225
concurrent transactions, 225–230
log-based recovery, 223–224
system model, 222–223
bounded-buffer problem, 205
critical-section problem, 193–195
hardware solution to, 197–200
Peterson's solution to, 195–197
dining-philosophers problem, 207–209, 212–214
examples of:
Java, 218
Linux, 221
Pthreads, 221–222
Solaris, 217–219
Windows XP, 220–221
monitors for, 209–217
dining-philosophers solution, 212–214
resumption of processes within, 215–217
semaphores, implementation using, 214–215
usage, 210–212
readers-writers problem, 206–207
semaphores for, 200–204
**process termination, deadlock recovery by,** 266
**production kernels (Linux),** 739
**profiles,** 719
**programs, processes vs.,** 82, 83. *See also* application programs
**program counters,** 21, 82
**program execution (operating system service),** 40
**program files,** 374
**program loading and execution,** 55
**programmable interval timer,** 509
**programmed I/O (PIO),** 353, 503
**programming-language support,** 55
**program threats,** 563–571
logic bombs, 565
stack- or buffer overflow attacks, 565–568
trap doors, 564–565
Trojan horses, 563–564
viruses, 568–571
**progressive download,** 716
**projects,** 176
**proportional allocation,** 341
**proportional share scheduling,** 708
**protection,** 531
access control for, 402–406
access matrix as model of, 538–542
control, access, 545–546
implementation, 542–545
capability-based systems, 547–550
Cambridge CAP system, 549–550
Hydra, 547–549
in computer systems, 26–27
domain of, 533–538
MULTICS example, 536–538
structure, 534–535
UNIX example, 535–536
error handling, 515
file, 374
of file systems, 402–407
goals of, 531–532
I/O, 515–516
language-based systems, 550–555
compiler-based enforcement, 550–553
Java, 553–555
as operating system service, 41
in paged environment, 295–296
permissions, 406
and principle of least privilege, 532–533
retrofitted, 407
and revocation of access rights, 546–547
security vs., 559
static vs. dynamic, 534
from viruses, 596–598
**protection domain,** 534
**protection mask (Linux),** 778
**protection subsystems (Windows XP),** 788
**protocols, Windows XP networking,** 822–824
**PTBR (page-table base register),** 293
**Pthreads,** 132–134
scheduling, 172–174
synchronization in, 221–222

**Pthread scheduling,** 708–710
**PTLR (page-table length register),** 296
**public domain,** 741
**public keys,** 580
**pull migration,** 170
**pure code,** 296
**pure demand paging,** 322
**push migration,** 170, 644

**Q**

**quantum,** 789
**queue(s),** 85–87
    capacity of, 102
    input, 278
    message, 848
    ready, 85, 87, 283
**queueing diagram,** 87
**queueing-network analysis,** 183

**R**

**race condition,** 193
**RAID (redundant arrays of inexpensive disks),** 468–477
    levels of, 470–476
    performance improvement, 470
    problems with, 477
    reliability improvement, 468–470
    structuring, 469
**RAID array,** 469
**RAID levels,** 470–474
**RAM (random-access memory),** 8
**random access,** 717
**random-access devices,** 506, 507, 844
**random-access memory (RAM),** 8
**random-access time (disks),** 452
**rate-monotonic scheduling algorithm,** 705–707
**raw disk,** 339, 416
**raw disk space,** 386
**raw I/O,** 508
**raw partitions,** 467
**RBAC (role-based access control),** 545
**RC 4000 operating system,** 848–849
**reaching algorithms,** 686–688
**read-ahead technique,** 435
**readers,** 206
**readers-writers problem,** 206–207
**reader-writer locks,** 207
**reading files,** 375
**read-modify-write cycle,** 473
**read only devices,** 506, 507
**read-only disks,** 480
**read-only memory (ROM),** 71, 463–464
**read queue,** 772
**read-write devices,** 506, 507
**read-write disks,** 479
**ready queue,** 85, 87, 283
**ready state,** 83
**ready thread state (Windows XP),** 789
**real-addressing mode,** 699
**real-time class,** 177
**real-time clients,** 728
**real-time operating systems,** 29–30
**real-time range (Linux schedulers),** 752
**real-time streaming,** 716, 726–728
**real-time systems,** 29–30, 695–696
    address translation in, 699–700
    characteristics of, 696–698
    CPU scheduling in, 704–710
    defined, 695
    features not needed in, 698–699
    footprint of, 697
    hard, 696, 722
    implementation of, 700–704
        and minimizing latency, 702–704
        and preemptive kernels, 701
        and priority-based scheduling, 700–701
    soft, 696, 722
    VxWorks example, 710–712
**real-time transport protocol (RTP),** 725
**real-time value (Linux),** 179
**reconfiguration,** 633
**records:**
    logical, 383
    master boot, 464
**recovery:**
    backup and restore, 436–437
    consistency checking, 435–436
    from deadlock, 266–267
        by process termination, 266
        by resource preemption, 267
    from failure, 633
    of files and directories, 435–437
    Windows XP, 816–817
**redirectors,** 826
**redundancy,** 469. *See also* RAID

**redundant arrays of inexpensive disks,** *see* RAID
**Reed-Solomon codes,** 473
**reentrant code (pure code),** 296
**reference bits,** 336
**Reference Model, ISO,** 585
**reference string,** 330
**register(s),** 47
  base, 276, 277
  limit, 276, 277
  memory-address, 279
  page-table base, 293
  page-table length, 296
  for page tables, 292–293
  relocation, 280
**registry,** 55, 810
**relative block number,** 383–384
**relative path names,** 390
**relative speed,** 194
**release() operation,** 377
**reliability,** 626
  of distributed operating systems, 612–613
  in multimedia systems, 721
  of Windows XP, 785
**relocation register,** 280
**remainder section,** 193
**remote file access (distributed file systems),** 646–651
  basic scheme for, 647
  and cache location, 647–648
  and cache-update policy, 648, 649
  and caching vs. remote service, 650–651
  and consistency, 649–650
**remote file systems,** 398
**remote file transfer,** 614–615
**remote login,** 614
**remote method invocation (RMI),** 114–115
**remote operations,** 443–444
**remote procedure calls (RPCs),** 825
**remote-service mechanism,** 646
**removable storage media,** 481–483
  application interface with, 481–482
  disks, 478–480
  and file naming, 482–483
  and hierarchical storage management, 483
  magnetic disks, 451–453
  magnetic tapes, 453–454, 480
**rendezvous,** 102
**repair, mean time to,** 469
**replay attacks,** 560
**replication,** 475
**repositioning (in files),** 375
**request edge,** 249
**request manager,** 772
**resident attributes,** 815
**resident monitor,** 841
**resolution:**
  name, 623
  and page size, 358
**resolving links,** 392
**resource allocation (operating system service),** 41
**resource-allocation graph algorithm,** 258–259
**resource allocator, operating system as,** 5
**resource fork,** 381
**resource manager,** 722
**resource preemption, deadlock recovery by,** 267
**resource-request algorithm,** 260–261
**resource reservations,** 721–722
**resource sharing,** 612
**resource utilization,** 4
**response time,** 16, 157–158
**restart area,** 817
**restore:**
  data, 436–437
  state, 89
**retrofitted protection mechanisms,** 407
**revocation of access rights,** 546–547
**rich text format (RTF),** 598
**rights amplification (Hydra),** 548
**ring algorithm,** 685–686
**ring structure,** 668
**risk assessment,** 592–593
**RMI,** *see* remote method invocation
**roaming profiles,** 827
**robotic jukebox,** 483
**robustness,** 631–633
**roles,** 545
**role-based access control (RBAC),** 545
**rolled-back transactions,** 223
**roll out, roll in,** 282
**ROM,** *see* read-only memory

**root partitions,** 417
**root uid (Linux),** 778
**rotational latency (disks),** 452, 457
**round-robin (RR) scheduling algorithm,** 164–166
**routing:**
and network communication, 625–626
in partially connected networks, 621–622
**routing protocols,** 626
**routing table,** 625
**RPCs (remote procedure calls)**
**RR scheduling algorithm,** *see* round-robin scheduling algorithm
**RSX operating system,** 853
**RTF (rich text format),** 598
**R-timestamp,** 229
**RTP (real-time transport protocol),** 725
**running state,** 83
**running system,** 72
**running thread state (Windows XP),** 789
**runqueue data structure,** 180, 752
**RW (read-write) format,** 24

## S

**safe computing,** 598
**safe sequence,** 256
**safety algorithm,** 260
**safety-critical systems,** 696
**sandbox (Tripwire file system),** 598
**SANs,** *see* storage-area networks
**SATA buses,** 453
**save, state,** 89
**scalability,** 634
**SCAN (elevator) scheduling algorithm,** 459–460, 724
**schedules,** 226
**scheduler(s),** 87–89
long-term, 88
medium-term, 89
short-term, 88
**scheduler activation,** 142–143
**scheduling:**
cooperative, 156
CPU, *see* CPU scheduling
disk scheduling algorithms, 456–462
C-SCAN, 460
FCFS, 457–458
LOOK, 460–461
SCAN, 459–460
selecting, 461–462
SSTF, 458–459
earliest-deadline-first, 707
I/O, 511–512
job, 17
in Linux, 751–756
kernel synchronization, 753–755
process, 751–753
symmetric multiprocessing, 755–756
nonpreemptive, 156
preemptive, 155–156
priority-based, 700–701
proportional share, 708
Pthread, 708–710
rate-monotonic, 705–707
thread, 172–173
in Windows XP, 789–790, 831–833
**scheduling rules,** 832
**SCOPE operating system,** 853
**script kiddies,** 568
**SCS (system-contention scope),** 172
**SCSI (small computer-systems interface),** 10
**SCSI buses,** 453
**SCSI initiator,** 455
**SCSI targets,** 455
**search path,** 389
**secondary memory,** 322
**secondary storage,** 9, 411. *See also* disk(s)
**second-chance page-replacement algorithm (clock algorithm),** 336–338
**second extended file system (ext2fs),** 766–769
**section objects,** 107
**sectors, disk,** 452
**sector slipping,** 465
**sector sparing,** 465, 820
**secure single sign-on,** 400
**secure systems,** 560

**security.** *See also* file access; program threats; protection; user authentication
classifications of, 600–602
in computer systems, 27
and firewalling, 599–600
implementation of, 592–599
and accounting, 599
and auditing, 599
and intrusion detection, 594–596
and logging, 599
and security policy, 592
and virus protection, 596–598
and vulnerability assessment, 592–594
levels of, 562
in Linux, 777–779
access control, 778–779
authentication, 777
as operating system service, 41
as problem, 559–563
protection vs., 559
and system/network threats, 571–576
denial of service, 575–576
port scanning, 575
worms, 572–575
use of cryptography for, 576–587
and encryption, 577–584
implementation, 584–585
SSL example, 585–587
via user authentication, 587–592
biometrics, 591–592
passwords, 588–591
Windows XP, 817–818
in Windows XP, 602–604, 785
**security access tokens (Windows XP),** 602
**security context (Windows XP),** 602–603
**security descriptor (Windows XP),** 603
**security domains,** 599
**security policy,** 592
**security reference monitor (SRM),** 808–809
**security-through-obscurity approach,** 594
**seeds,** 590–591
**seek, file,** 375
**seek time (disks),** 452, 457
**segmentation,** 302–305
basic method, 302–304
defined, 303
hardware, 304–305
Intel Pentium example, 305–307
**segment base,** 304
**segment limit,** 304
**segment tables,** 304
**semantics:**
consistency, 401–402
copy, 513
immutable-shared-files, 402
session, 402
**semaphore(s),** 200–204
binary, 201
counting, 201
and deadlocks, 204
defined, 200
implementation, 202–204
implementation of monitors using, 214–215
and starvation, 204
usage of, 201
Windows XP, 790
**semiconductor memory,** 10
**sense key,** 515
**sequential access (files),** 382–383
**sequential-access devices,** 844
**sequential devices,** 506, 507
**serial ATA (SATA) buses,** 453
**serializability,** 225–227
**serial schedule,** 226
**server(s),** 5
cluster, 655
defined, 642
in SSL, 586
**server-message-block (SMB),** 822–823
**server subject (Windows XP),** 603
**services, operating system,** 39–41
**session hijacking,** 561
**session layer,** 629
**session object,** 798
**session semantics,** 402
**session space,** 797
**sharable devices,** 506, 507
**shares,** 176
**shared files, immutable,** 402
**shared libraries,** 281–282, 318
**shared lock,** 378
**shared lock mode,** 672

**shared memory,** 96, 318
**shared-memory model,** 54, 97–99
**shared name space,** 655
**sharing:**
load, 169, 612
and paging, 296–297
resource, 612
time, 16
**shells,** 41, 121–123
**shell script,** 379
**shortest-job-first (SJF) scheduling algorithm,** 159–162
**shortest-remaining-time-first scheduling,** 162
**shortest-seek-time (SSTF) scheduling algorithm,** 458–459
**short-term scheduler (CPU scheduler),** 88, 155
**shoulder surfing,** 588
**signals:**
Linux, 773
UNIX, 123, 139–141
**signaled state,** 220
**signal handlers,** 139–141
**signal-safe functions,** 123–124
**signatures,** 595
**signature-based detection,** 595
**simple operating system structure,** 58–59
**simple subject (Windows XP),** 602
**simulations,** 183–184
**single indirect blocks,** 427
**single-level directories,** 387
**single-processor systems,** 12–14, 153
**single-threaded processes,** 127
**SJF scheduling algorithm,** *see* shortest-job-first scheduling algorithm
**skeleton,** 114
**slab allocation,** 355–356, 758
**Sleeping-Barber Problem,** 233
**slices,** 386
**small-area networks,** 28
**small computer-systems interface,** *see under* SCSI
**SMB,** *see* server-message-block
**SMP,** *see* symmetric multiprocessing
**sniffing,** 588
**social engineering,** 562
**sockets,** 108–111
**socket interface,** 508
**SOC strategy,** *see* system-on-chip strategy
**soft affinity,** 170
**soft error,** 463
**soft real-time systems,** 696, 722
**software capability,** 549
**software interrupts (traps),** 502
**software objects,** 533
**Solaris:**
scheduling example, 173, 175–177
swap-space management in, 467
synchronization in, 217–219
virtual memory in, 363–365
**Solaris 10 Dynamic Tracing Facility,** 52
**solid-state disks,** 24
**sorted queue,** 772
**source-code viruses,** 570
**source files,** 374
**sparseness,** 300, 318
**special-purpose computer systems,** 29–31
handheld systems, 30–31
multimedia systems, 30
real-time embedded systems, 29–30
**speed, relative,** 194
**speed of operations:**
for I/O devices, 506, 507
**spinlock,** 202
**spoofed client identification,** 398
**spoofing,** 599
**spool,** 514
**spooling,** 514–515, 844–845
**spyware,** 564
**SRM,** *see* security reference monitor
**SSL 3.0,** 585–587
**SSTF scheduling algorithm,** *see* shortest-seek-time scheduling algorithm
**stable storage,** 223, 477–478
**stack,** 47, 82
**stack algorithms,** 335
**stack frame,** 566–567
**stack inspection,** 554
**stack-overflow attacks,** 565–568
**stage (magnetic tape),** 480
**stalling,** 276
**standby thread state (Windows XP),** 789
**starvation,** *see* indefinite blocking
**state (of process),** 83
**stateful file service,** 651
**state information,** 40–401
**stateless DFS,** 401
**stateless file service,** 651

**stateless protocols,** 727
**state restore,** 89
**state save,** 89
**static linking,** 281–282, 764
**static priority,** 722
**static protection,** 534
**status information,** 55
**status register,** 498
**stealth viruses,** 570
**storage.** *See also* mass-storage structure
  holographic, 480
  nonvolatile, 10, 223
  secondary, 9, 411
  stable, 223
  tertiary, 24
  utility, 476
  volatile, 10, 223
**storage-area networks (SANs),** 15, 455, 456
**storage array,** 469
**storage management,** 22–26
  caching, 24–26
  I/O systems, 26
  mass-storage management, 23–24
**stream ciphers,** 579–580
**stream head,** 520
**streaming,** 716–717
**stream modules,** 520
**STREAMS mechanism,** 520–522
**string, reference,** 330
**stripe set,** 818–820
**stubs,** 114, 281
**stub routines,** 825
**superblock,** 414
**superblock objects,** 419, 765
**supervisor mode,** *see* kernel mode
**suspended state,** 832
**sustained bandwidth,** 484
**swap map,** 468
**swapper (term),** 319
**swapping,** 17, 89, 282–284, 319
  in Linux, 761
  paging vs., 466
**swap space,** 322
**swap-space management,** 466–468
**switch architecture,** 11
**switching:**
  circuit, 626–627
  domain, 535
  message, 627
  packet, 627
**symbolic links,** 794
**symbolic-link objects,** 794
**symmetric encryption,** 579–580
**symmetric mode,** 15
**symmetric multiprocessing (SMP),** 13–14, 169, 171–172, 755–756
**synchronization,** 101–102. *See also* process synchronization
**synchronous devices,** 506, 507
**synchronous message passing,** 102
**synchronous writes,** 434
**SYSGEN,** *see* system generation
**system boot,** 71–72
**system calls (monitor calls),** 7, 43–55
  and API, 44–46
  for communication, 54–55
  for device management, 53
  for file management, 53
  functioning of, 43–44
  for information maintenance, 53–54
  for process control, 47–52
**system-call firewalls,** 600
**system-call interface,** 46
**system-contention scope (SCS),** 172
**system device,** 810
**system disk,** *see* boot disk
**system files,** 389
**system generation (SYSGEN),** 70–71
**system hive,** 810
**system libraries (Linux),** 743, 744
**system mode,** *see* kernel mode
**system-on-chip (SOC) strategy,** 697, 698
**system process (Windows XP),** 810
**system programs,** 55–56
**system resource-allocation graph,** 249–251
**system restore,** 810
**systems layer,** 719
**system utilities,** 55–56, 743–744
**system-wide open-file table,** 414

## T

**table(s),** 316
  file-allocation, 425
  hash, 420
  master file, 414

mount, 417, 518
object, 796
open-file, 376
page, 322, 799
per-process open-file, 414
routing, 625
segment, 304
system-wide open-file, 414
**tags,** 543
**tapes, magnetic,** 453–454, 480
**target thread,** 139
**tasks:**
Linux, 750–751
VxWorks, 710
**task control blocks,** *see* process control blocks
**TCB (trusted computer base),** 601
**TCP/IP,** *see* Transmission Control Protocol/Internet Protocol
**TCP sockets,** 109
**TDI (transport driver interface),** 822
**telnet,** 614
**Tenex operating system,** 853
**terminal concentrators,** 523
**terminated state,** 83
**terminated thread state (Windows XP),** 789
**termination:**
cascading, 95
process, 90–95, 266
**tertiary-storage,** 478–488
future technology for, 480
and operating system support, 480–483
performance issues with, 484–488
removable disks, 478–480
tapes, 480
**tertiary storage devices,** 24
**text files,** 374
**text section (of process),** 82
**theft of service,** 560
**THE operating system,** 846–848
**thrashing,** 343–348
cause of, 343–345
defined, 343
and page-fault-frequency strategy, 347–348
and working-set model, 345–347
**threads.** *See also* multithreading
cancellation, thread, 139
components of, 127
functions of, 127–129
idle, 177
kernel, 129
in Linux, 144–146, 750–751
pools, thread, 141–142
and process model, 84–85
scheduling of, 172–173
target, 139
user, 129
in Windows XP, 144, 145, 789–790, 830, 832–833
**thread libraries,** 131–138
about, 131–132
Java threads, 134–138
Pthreads, 132–134
Win32 threads, 134
**thread pool,** 832
**thread scheduling,** 153
**thread-specific data,** 142
**threats,** 560. *See also* program threats
**throughput,** 157, 720
**thunking,** 812
**tightly coupled systems,** *see* multiprocessor systems
**time:**
compile, 278
effective access, 323
effective memory-access, 294
execution, 278
of file creation/use, 375
load, 278
response, 16, 157–158
turnaround, 157
waiting, 157
**time-out schemes,** 632, 686–687
**time quantum,** 164
**timer:**
programmable interval, 509
variable, 20
**timers,** 509–510
**timer objects,** 790
**time sharing (multitasking),** 16
**timestamp-based protocols,** 228–230
**timestamping,** 675–676
**timestamps,** 665
**TLB,** *see* translation look-aside buffer

**TLB miss,** 293
**TLB reach,** 358–359
**tokens,** 628, 668
**token passing,** 628, 668
**top half interrupt service routines,** 755
**topology, network,** 620–622
**Torvalds, Linus,** 737
**trace tapes,** 184
**tracks, disk,** 452
**traditional computing,** 31–32
**transactions,** 222. *See also* atomic transactions
  defined, 768
  in Linux, 768–769
  in log-structured file systems, 437–438
**Transarc DFS,** 654
**transfer rate (disks),** 452, 453
**transition thread state (Windows XP),** 789
**transitive trust,** 828
**translation coordinator,** 669
**translation look-aside buffer (TLB),** 293, 800
**transmission control protocol (TCP),** 631
**Transmission Control Protocol/Internet Protocol (TCP/IP),** 823
**transparency,** 633–634, 642, 643
**transport driver interface (TDI),** 822
**transport layer,** 629
**transport-layer protocol (TCP),** 584
**traps,** 18, 321, 502
**trap doors,** 564–565
**tree-structured directories,** 389–391
**triple DES,** 579
**triple indirect blocks,** 427
**Tripwire file system,** 597–598
**Trojan horses,** 563–564
**trusted computer base (TCB),** 601
**trust relationships,** 828
**tunneling viruses,** 571
**turnaround time,** 157
**turnstiles,** 219
**two-factor authentication,** 591
**twofish algorithm,** 579
**two-level directories,** 388–389
**two-phase commit (2PC) protocol,** 669–672
**two-phase locking protocol,** 228
**two tuple,** 303
**type safety (Java),** 555

U

**UDP (user datagram protocol),** 631
**UDP sockets,** 109
**UFD (user file directory),** 388
**UFS (UNIX file system),** 413
**UI,** *see* user interface
**unbounded capacity (of queue),** 102
**UNC (uniform naming convention),** 824
**unformatted disk space,** 386
**unicasting,** 725
**UNICODE,** 787
**unified buffer cache,** 433, 434
**unified virtual memory,** 433
**uniform naming convention (UNC),** 824
**universal serial buses (USBs),** 453
**UNIX file system (UFS),** 413
**UNIX operating system:**
  consistency semantics for, 401
  domain switching in, 535–536
  and Linux, 737
  permissions in, 406
  shell and history feature (project), 121–125
  signals in, 123, 139–141
  swapping in, 284
**unreliability,** 626
**unreliable communications,** 686–687
**upcalls,** 143
**upcall handler,** 143
**USBs,** *see* universal serial buses
**used objects,** 356, 759
**users,** 4–5, 397–398
**user accounts,** 602
**user authentication,** 587–592
  with biometrics, 591–592
  with passwords, 588–591
**user datagram protocol (UDP),** 631
**user-defined signal handlers,** 140
**user file directory (UFD),** 388
**user identifiers (user IDs),** 27
  effective, 27
  for files, 375
**user interface (UI),** 40–43
**user mobility,** 440
**user mode,** 18
**user programs (user tasks),** 81, 762–763
**user rights (Linux),** 778

**user threads,** 129
**utility storage,** 476
**utilization,** 840

**V**

**VACB,** *see* virtual address control block
**VADs (virtual address descriptors),** 802
**valid-invalid bit,** 295
**variable class,** 177
**variables, automatic,** 566
**variable timer,** 20
**VDM,** *see* virtual DOS machine
**vector programs,** 573
**vfork() (virtual memory fork),** 327
**VFS,** *see* virtual file system
**victim frames,** 329
**views,** 798
**virtual address,** 279
**virtual address control block (VACB),** 806, 807
**virtual address descriptors (VADs),** 802
**virtual address space,** 317, 760–761
**virtual DOS machine (VDM),** 811–812
**virtual file system (VFS),** 417–419, 765–766
**virtual machines,** 64–69
  basic idea of, 64
  benefits of, 66
  implementation of, 65–66
  Java Virtual Machine as example of, 68
  VMware as example of, 67
**virtual memory,** 17, 315–318
  and copy-on-write technique, 325–327
  demand paging for conserving, 319–325
    basic mechanism, 320–322
    with inverted page tables, 359–360
    and I/O interlock, 361–362
    and page size, 357–358
    and performance, 323–325
    and prepaging, 357
    and program structure, 360–361
    pure demand paging, 322
    and restarting instructions, 322–323
    and TLB reach, 358–359
  direct virtual memory access, 504
  and frame allocation, 340–343
    equal allocation, 341
    global vs. local allocation, 342–343
    proportional allocation, 341–342
  kernel, 762
  and kernel memory allocation, 353–356
  in Linux, 759–762
  and memory mapping, 348–353
    basic mechanism, 348–350
    I/O, memory-mapped, 353
    in Win32 API, 350–353
  network, 647
  page replacement for conserving, 327–339
    and application performance, 339
    basic mechanism, 328–331
    counting-based page replacement, 338
    FIFO page replacement, 331–333
    LRU-approximation page replacement, 336–338
    LRU page replacement, 334–336
    optimal page replacement, 332–334
    and page-buffering algorithms, 338–339
  separation of logical memory from physical memory by, 317
  size of, 316
  in Solaris, 363–365
  and thrashing, 343–348
    cause, 343–345
    page-fault-frequency strategy, 347–348
    working-set model, 345–347
  unified, 433
  in Windows XP, 363
**virtual memory fork,** 327
**virtual memory (VM) manager,** 796–802
**virtual memory regions,** 760

**virtual private networks (VPNs),** 585, 823
**virtual routing,** 625
**viruses,** 568–571, 596–598
**virus dropper,** 569
**VM manager,** *see* virtual memory manager
**VMS operating system,** 853
**VMware,** 67
**vnode,** 418
**vnode number (NFS V4),** 656
**volatile storage,** 10, 223
**volumes,** 386, 656
**volume control block,** 414
**volume-location database (NFS V4),** 656
**volume management (Windows XP),** 818–821
**volume set,** 818
**volume shadow copies,** 821–822
**volume table of contents,** 386
**von Neumann architecture,** 8
**VPNs,** *see* virtual private networks
**vulnerability scans,** 592–593
**VxWorks,** 710–712

**W**

**WAFL file system,** 444–446
**wait-die scheme,** 677–678
**waiting state,** 83
**waiting thread state (Windows XP),** 789
**waiting time,** 157
**wait queue,** 773
**WANs,** *see* wide-area networks
**Web-based computing,** 34
**web clipping,** 31
**Web distributed authoring and versioning (WebDAV),** 824
**wide-area networks (WANs),** 15, 28, 619–620
**Win32 API,** 350–353, 783–784, 813
**Win32 thread library,** 134
**Windows, swapping in,** 284
**Windows 2000,** 785, 787
**Windows NT,** 783–784
**Windows XP,** 783–836
  application compatibility of, 785–786
  design principles for, 785–787
  desktop versions of, 784
  environmental subsystems for, 811–814
    16-bit Windows, 812
    32-bit Windows, 812–813
    logon, 814
    MS-DOS, 811–812
    POSIX, 813–814
    security, 814
    Win32, 813
  extensibility of, 786–787
  file systems, 814–822
    change journal, 821
    compression and encryption, 821
    mount points, 821
    NTFS B+ tree, 816
    NTFS internal layout, 814–816
    NTFS metadata, 816
    recovery, 816–817
    security, 817–818
    volume management and fault tolerance, 818–821
    volume shadow copies, 821–822
  history of, 783–785
  interprocess communication example, 106–108
  networking, 822–829
    Active Directory, 828
    distributed-processing mechanisms, 824–826
    domains, 827–828
    interfaces, 822
    name resolution, 828–829
    protocols, 822–824
    redirectors and servers, 826–827
  performance of, 786
  portability of, 787
  programmer interface, 829–836
    interprocess communication, 833–834
    kernel object access, 829
    memory management, 834–836
    process management, 830–833
    sharing objects between processes, 829–830
  reliability of, 785

scheduling example, 177–179
security, 814
security in, 785
synchronization in, 220–221
system components for, 787–811
executive, *see* Windows XP executive
hardware-abstraction layer, 788
kernel, 788–793
threads example, 144, 145
virtual memory in, 363
**Windows XP executive,** 793–811
booting, 810–811
cache manager, 806–808
I/O manager, 805–806
local procedure call facility, 804–805
object manager, 793–796
plug-and-play and power managers, 809–810
process manager, 802–804
registry, 810
security reference monitor, 808–809
virtual memory manager, 796–802
**Winsock,** 824
**wireless networks,** 31
**Witness,** 255–256
**working sets,** 346, 348
**working-set maximum (Windows XP),** 363
**working-set minimum (Windows XP),** 363
**working-set model,** 345–347
**workstations,** 5
**world rights (Linux),** 778
**World Wide Web,** 398
**worms,** 572–575
**WORM disks,** *see* write-once, read-many-times disks
**WORM (write-once, read-many) format,** 24
**worst-fit strategy,** 287
**wound-wait scheme,** 677–678
**write-ahead logging,** 223–224
**write-back caching,** 648
**write-on-close policy,** 649
**write-once, read-many-times (WORM) disks,** 479
**write only devices,** 506, 507
**write queue,** 772
**writers,** 206
**write-through policy,** 648
**writing files,** 375
**W-timestamp,** 229

## X

**XDR (external data representation),** 112
**XDS-940 operating system,** 846–847
**Xerox,** 42
**XML firewalls,** 600

## Z

**zero capacity (of queue),** 102
**zero-day attacks,** 595
**zero-fill-on-demand technique, 327**
**zipped files,** 719
**zombie systems,** 575
**zones (Linux),** 756

## Another defining moment in the evolution of operating systems

Small footprint operating systems, such as those driving the handheld devices that the baby dinosaurs are using on the cover, are just one of the cutting-edge applications you'll find in Silberschatz, Galvin, and Gagne's *Operating System Concepts, Seventh Edition.*

By staying current, remaining relevant, and adapting to emerging course needs, this market-leading text has continued to define the operating systems course. This Seventh Edition not only presents the latest and most relevant systems, it also digs deeper to uncover those fundamental concepts that have remained constant throughout the evolution of today's operating systems. With this strong conceptual foundation in place, students can more easily understand the details related to specific systems.

### New Adaptations

- Increased coverage of user perspective in Chapter 1.
- Increased coverage of OS design throughout.
- A new chapter on real-time and embedded systems (Chapter 19).
- A new chapter on multimedia (Chapter 20).
- Additional coverage of security and protection.
- Additional coverage of distributed programming.
- New exercises at the end of each chapter.
- New programming assignments and projects at the end of each chapter.
- New student-focused pedagogy and a new two-color design to enhance the learning process.

### The next evolutionary leap in the classroom—eGrade Plus!

Instructors, looking for a better way to create and assign homework, prepare class presentations, and manage your course? *eGrade Plus* provides an integrated suite of teaching and learning resources, along with a complete online version of the text, in one easy-to-use web site. Go to www.wiley.com/college/egradeplus for more information.

See why the most widely used operating systems text stays that way.

ISBN 0-471-69466-5
9 780471 694663 90000

WILEY

www.wiley.com/college/silberschatz

SEVENTH EDITION

SILBERSCHATZ • GALVIN • GAGNE

*Operating System Concepts*

WILEY

# Part One

# Overview

An *operating system* acts as an intermediary between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a *convenient* and *efficient* manner.

An operating system is software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well delineated portion of the system, with carefully defined inputs, outputs, and functions.




# Introduction

An **operating system** is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how varied they are in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Operating systems for handheld computers are designed to provide an environment in which a user can easily interface with the computer to execute programs. Thus, some operating systems are designed to be *convenient*, others to be *efficient*, and others some combination of the two.

Before we can explore the details of computer system operation, we need to know something about system structure. We begin by discussing the basic functions of system startup, I/O, and storage. We also describe the basic computer architecture that makes it possible to write a functional operating system.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. In this chapter we provide a general overview of the major components of an operating system.

## CHAPTER OBJECTIVES

- To provide a grand tour of the major operating systems components.
- To provide coverage of basic computer system organization.

## 1.1 What Operating Systems Do

We begin our discussion by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into four components: the *hardware*, the *operating system*, the *application programs*, and the *users* (Figure 1.1).

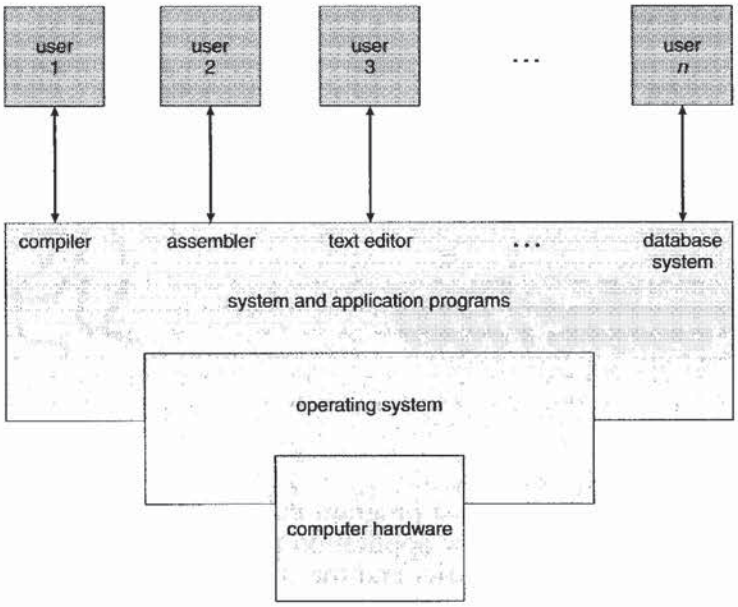



**Figure 1.1** Abstract view of the components of a computer system.

The **hardware**—the **central processing unit (CPU)**, the **memory**, and the **input/output (I/O) devices**—provides the basic computing resources for the system. The **application programs**—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls and coordinates the use of the hardware among the various application programs for the various users.

We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a *government*. Like a government, it performs no useful function by itself. It simply provides an *environment* within which other programs can do useful work.

To understand more fully the operating system's role, we next explore operating systems from two viewpoints: that of the user and that of the system.

### 1.1.1 User View

The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and none paid to **resource utilization**—how various hardware and software resources are shared. Performance is, of course, important to the user; but rather than resource utilization, such systems are optimized for the single-user experience.

In other cases, a user sits at a terminal connected to a **mainframe** or **minicomputer**. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization—to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share.

In still other cases, users sit at **workstations** connected to networks of other workstations and **servers**. These users have dedicated resources at their disposal, but they also share resources such as networking and servers—file, compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.

Recently, many varieties of handheld computers have come into fashion. Most of these devices are standalone units for individual users. Some are connected to networks, either directly by wire or (more often) through wireless modems and networking. Because of power, speed, and interface limitations, they perform relatively few remote operations. Their operating systems are designed mostly for individual usability, but performance per amount of battery life is important as well.

Some computers have little or no user view. For example, embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention.

### 1.1.2 System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly. As we have seen, resource allocation is especially important where many users access the same mainframe or minicomputer.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

### 1.1.3 Defining Operating Systems

We have looked at the operating system's role from the views of the user and of the system. How, though, can we define what an operating system is? In general, we have no completely adequate definition of an operating system. Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Toward this goal, computer hardware is constructed. Since bare hardware alone is not particularly easy to use, application programs are

developed. These programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

In addition, we have no universally accepted definition of what is part of the operating system. A simple viewpoint is that it includes everything a vendor ships when you order "the operating system." The features included, however, vary greatly across systems. Some systems take up less than 1 megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are entirely based on graphical windowing systems. (A kilobyte, or KB, is 1,024 bytes; a megabyte, or MB, is $1{,}024^2$ bytes; and a gigabyte, or GB, is $1{,}024^3$ bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes.) A more common definition is that the operating system is the one program running at all times on the computer (usually called the **kernel**), with all else being systems programs and application programs. This last definition is the one that we generally follow.

The matter of what constitutes an operating system has become increasingly important. In 1998, the United States Department of Justice filed suit against Microsoft, in essence claiming that Microsoft included too much functionality in its operating systems and thus prevented application vendors from competing. For example, a web browser was an integral part of the operating system. As a result, Microsoft was found guilty of using its operating system monopoly to limit competition.

## 1.2 Computer-System Organization

Before we can explore the details of how computer systems operate, we need a general knowledge of the structure of a computer system. In this section, we look at several parts of this structure to round out our background knowledge. The section is mostly concerned with computer-system organization, so you can skim or skip it if you already understand the concepts.

### 1.2.1 Computer-System Operation

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, and video displays). The CPU and the device controllers can execute concurrently, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, it is stored in read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM), known by the general term **firmware**, within the computer hardware. It initializes all aspects of the system, from CPU registers to device

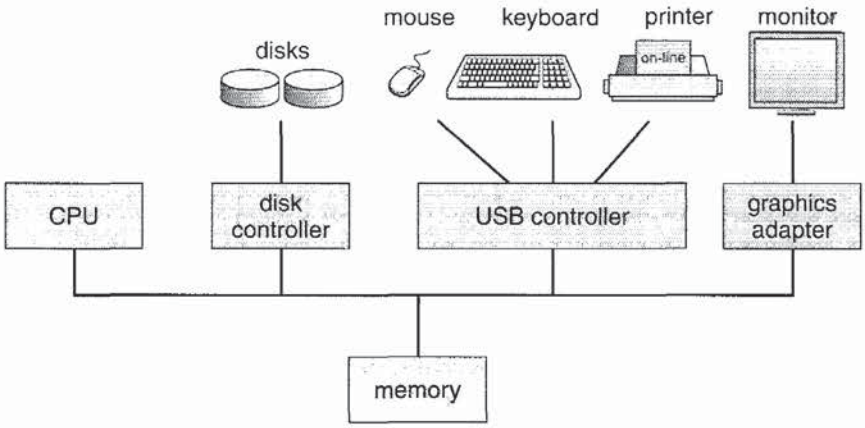



**Figure 1.2** A modern computer system.

controllers to memory contents. The bootstrap program must know how to load the operating system and to start executing that system. To accomplish this goal, the bootstrap program must locate and load into memory the operating-system kernel. The operating system then starts executing the first process, such as "init," and waits for some event to occur.

The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**).

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A time line of this operation is shown in Figure 1.3.

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine.

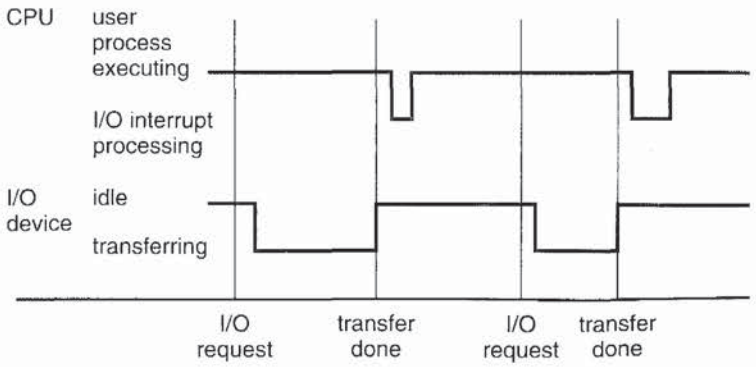



**Figure 1.3** Interrupt time line for a single process doing output.

The straightforward method for handling this transfer would be to invoke a generic routine to examine the interrupt information; the routine, in turn, would call the interrupt-specific handler. However, interrupts must be handled quickly. Since only a predefined number of interrupts is possible, a table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first 100 or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector**, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as Windows and UNIX dispatch interrupts in this manner.

The interrupt architecture must also save the address of the interrupted instruction. Many old designs simply stored the interrupt address in a fixed location or in a location indexed by the device number. More recent architectures store the return address on the system stack. If the interrupt routine needs to modify the processor state—for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

### 1.2.2 Storage Structure

Computer programs must be in main memory (also called **random-access memory** or RAM) to be executed. Main memory is the only large storage area (millions to billions of bytes) that the processor can access directly. It commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM)**, which forms an array of memory words. Each word has its own address. Interaction is achieved through a sequence of `load` or `store` instructions to specific memory addresses. The `load` instruction moves a word from main memory to an internal register within the CPU, whereas the `store` instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution.

A typical instruction–execution cycle, as executed on a system with a **von Neumann** architecture, first fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore *how* a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible for the following two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory is a *volatile* storage device that loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The most common secondary-storage device is a **magnetic disk**, which provides storage for both programs and data. Most programs (web browsers, compilers, word processors, spreadsheets, and so on) are stored on a disk until they are loaded into memory. Many programs then use the disk as both a source and a destination of the information for their processing. Hence, the proper management of disk storage is of central importance to a computer system, as we discuss in Chapter 12.

In a larger sense, however, the storage structure that we have described—consisting of registers, main memory, and magnetic disks—is only one of many possible storage systems. Others include cache memory, CD-ROM, magnetic tapes, and so on. Each storage system provides the basic functions of storing a datum and of holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, cost, size, and volatility.

The wide variety of storage systems in a computer system can be organized in a hierarchy (Figure 1.4) according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit

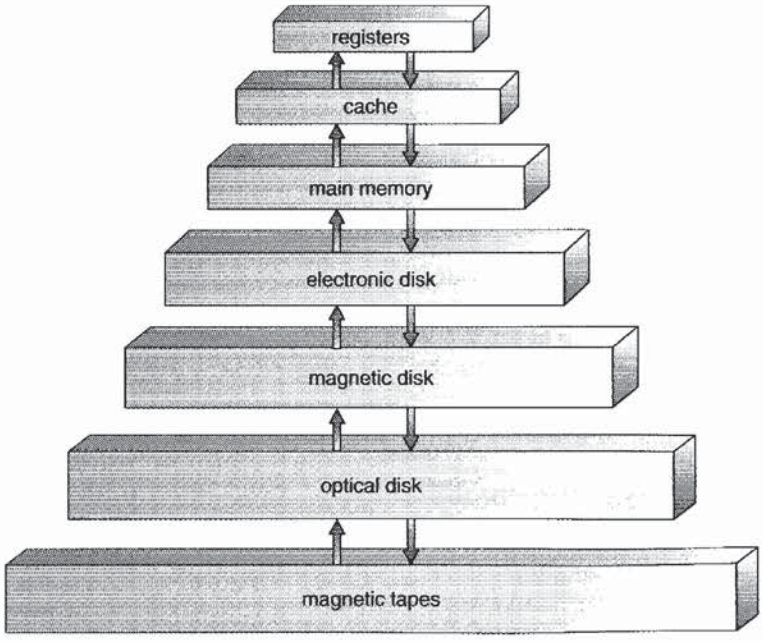



**Figure 1.4** Storage-device hierarchy.

generally decreases, whereas the access time generally increases. This trade-off is reasonable; if a given storage system were both faster and less expensive than another—other properties being the same—then there would be no reason to use the slower, more expensive memory. In fact, many early storage devices, including paper tape and core memories, are relegated to museums now that magnetic tape and **semiconductor memory** have become faster and cheaper. The top four levels of memory in Figure 1.4 may be constructed using semiconductor memory.

In addition to differing in speed and cost, the various storage systems are either volatile or nonvolatile. As mentioned earlier, **volatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safekeeping. In the hierarchy shown in Figure 1.4, the storage systems above the electronic disk are volatile, whereas those below are nonvolatile. An **electronic disk** can be designed to be either volatile or nonvolatile. During normal operation, the electronic disk stores data in a large DRAM array, which is volatile. But many electronic-disk devices contain a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, the electronic-disk controller copies the data from RAM to the magnetic disk. When external power is restored, the controller copies the data back into the RAM. Another form of electronic disk is flash memory, which is popular in cameras and **personal digital assistants (PDAs)**, in robots, and increasingly as removable storage on general-purpose computers. Flash memory is slower than DRAM but needs no power to retain its contents. Another form of nonvolatile storage is **NVRAM**, which is DRAM with battery backup power. This memory can be as fast as DRAM but has a limited duration in which it is nonvolatile.

The design of a complete memory system must balance all the factors just discussed: It must use only as much expensive memory as necessary while providing as much inexpensive, nonvolatile memory as possible. Caches can be installed to improve performance where a large access-time or transfer-rate disparity exists between two components.

### 1.2.3 I/O Structure

Storage is only one of many types of I/O devices within a computer. A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices. Therefore, we now provide an overview of I/O.

A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device. Depending on the controller, there may be more than one attached device. For instance, seven or more devices can be attached to the **small computer-systems interface (SCSI)** controller. A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage. Typically, operating systems have a **device driver** for each device controller. This device

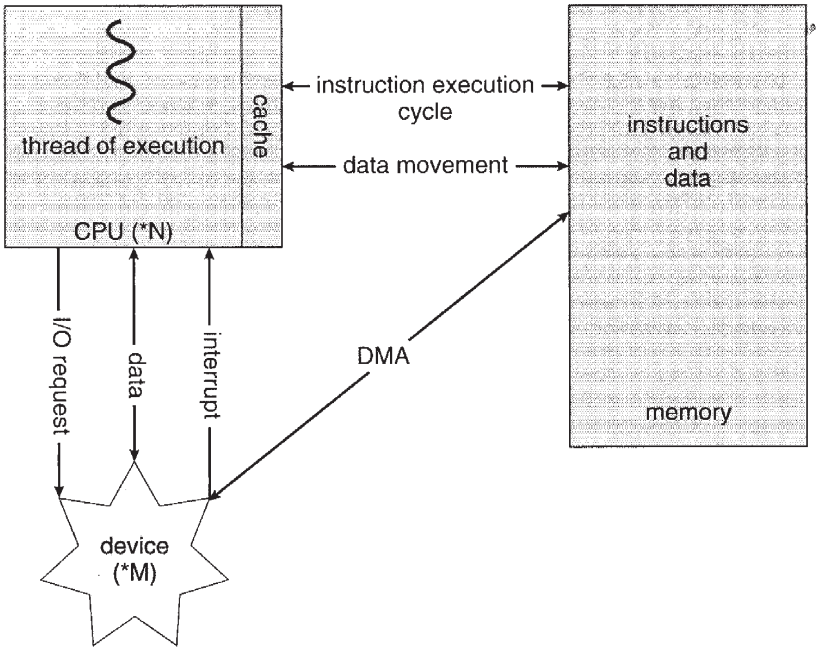



**Figure 1.5** How a modern computer system works.

driver understands the device controller and presents a uniform interface to the device to the rest of the operating system.

To start an I/O operation, the device driver loads the appropriate registers within the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take (such as "read a character from the keyboard"). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation. The device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read. For other operations, the device driver returns status information.

This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, **direct memory access (DMA)** is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective. Figure 1.5 shows the interplay of all components of a computer system.

## 1.3 Computer-System Architecture

In Section 1.2 we introduced the general structure of a typical computer system. A computer system may be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

### 1.3.1 Single-Processor Systems

Most systems use a single processor. The variety of single-processor systems may be surprising, however, since these systems range from PDAs through mainframes. On a single-processor system, there is one main CPU capable of executing a general-purpose instruction set, including instructions from user processes. Almost all systems have other special-purpose processors as well. They may come in the form of device-specific processors, such as disk, keyboard, and graphics controllers; or, on mainframes, they may come in the form of more general-purpose processors, such as I/O processors that move data rapidly among the components of the system.

All of these special-purpose processors run a limited instruction set and do not run user processes. Sometimes they are managed by the operating system, in that the operating system sends them information about their next task and monitors their status. For example, a disk-controller microprocessor receives a sequence of requests from the main CPU and implements its own disk queue and scheduling algorithm. This arrangement relieves the main CPU of the overhead of disk scheduling. PCs contain a microprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In other systems or circumstances, special-purpose processors are low-level components built into the hardware. The operating system cannot communicate with these processors; they do their jobs autonomously. The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU, then the system is a single-processor system.

### 1.3.2 Multiprocessor Systems

Although single-processor systems are most common, **multiprocessor systems** (also known as **parallel systems** or **tightly coupled systems**) are growing in importance. Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices.

Multiprocessor systems have three main advantages:

1. **Increased throughput**. By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with $N$ processors is not $N$, however; rather, it is less than $N$. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors. Similarly, $N$ programmers working closely together do not produce $N$ times the amount of work a single programmer would produce.

2. **Economy of scale**. Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.

3. **Increased reliability**. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs only 10 percent slower, rather than failing altogether.

Increased reliability of a computer system is crucial in many applications. The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation. Note that fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected. The HP NonStop system (formerly Tandem) system uses both hardware and software duplication to ensure continued operation despite faults. The system consists of multiple pairs of CPUs, working in lockstep. Both processors in the pair execute each instruction and compare the results. If the results differ, then one CPU of the pair is at fault, and both are halted. The process that was being executed is then moved to another pair of CPUs, and the instruction that failed is restarted. This solution is expensive, since it involves special hardware and considerable hardware duplication.

The multiple-processor systems in use today are of two types. Some systems use **asymmetric multiprocessing**, in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master–slave relationship. The master processor schedules and allocates work to the slave processors.

The most common systems use **symmetric multiprocessing (SMP)**, in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no master–slave relationship exists between processors. Figure 1.6 illustrates a typical SMP architecture. An example of the SMP system is Solaris, a commercial version of UNIX designed by Sun Microsystems. A Solaris system can be configured to employ dozens of processors, all running Solaris. The benefit of this model is that many processes

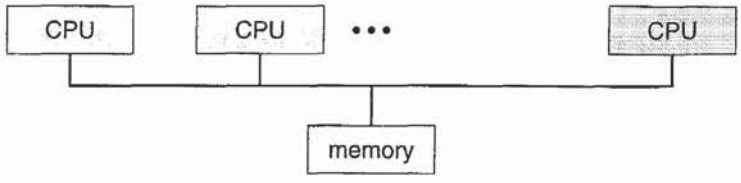



**Figure 1.6** Symmetric multiprocessing architecture.

can run simultaneously—$N$ processes can run if there are $N$ CPUs—without causing a significant deterioration of performance. However, we must carefully control I/O to ensure that the data reach the appropriate processor. Also, since the CPUs are separate, one may be sitting idle while another is overloaded, resulting in inefficiencies. These inefficiencies can be avoided if the processors share certain data structures. A multiprocessor system of this form will allow processes and resources—such as memory—to be shared dynamically among the various processors and can lower the variance among the processors. Such a system must be written carefully, as we shall see in Chapter 6. Virtually all modern operating systems—including Windows, Windows XP, Mac OS X, and Linux—now provide support for SMP.

The difference between symmetric and asymmetric multiprocessing may result from either hardware or software. Special hardware can differentiate the multiple processors, or the software can be written to allow only one master and multiple slaves. For instance, Sun's operating system SunOS Version 4 provided asymmetric multiprocessing, whereas Version 5 (Solaris) is symmetric on the same hardware.

A recent trend in CPU design is to include multiple compute **cores** on a single chip. In essence, these are multiprocessor chips. Two-way chips are becoming mainstream, while $N$-way chips are going to be common in high-end systems. Aside from architectural considerations such as cache, memory, and bus contention, these multi-core CPUs look to the operating system just as $N$ standard processors.

Lastly, **blade servers** are a recent development in which multiple processor boards, I/O boards, and networking boards are placed in the same chassis. The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system. Some blade-server boards are multiprocessor as well, which blurs the lines between types of computers. In essence, those servers consist of multiple independent multiprocessor systems.

### 1.3.3 Clustered Systems

Another type of multiple-CPU system is the **clustered system**. Like multiprocessor systems, clustered systems gather together multiple CPUs to accomplish computational work. Clustered systems differ from multiprocessor systems, however, in that they are composed of two or more individual systems coupled together. The definition of the term *clustered* is not concrete; many commercial packages wrestle with what a clustered system is and why one form is better than another. The generally accepted definition is that clustered computers share storage and are closely linked via a **local-area network (LAN)** (as described in Section 1.10) or a faster interconnect such as InfiniBand.

Clustering is usually used to provide **high-availability** service; that is, service will continue even if one or more systems in the cluster fail. High availability is generally obtained by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over the LAN). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the applications that were running on the failed machine. The users and clients of the applications see only a brief interruption of service.

Clustering can be structured asymmetrically or symmetrically. In **asymmetric clustering**, one machine is in **hot-standby mode** while the other is running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server. In **symmetric mode**, two or more hosts are running applications, and are monitoring each other. This mode is obviously more efficient, as it uses all of the available hardware. It does require that more than one application be available to run.

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN) (as described in Section 1.10). Parallel clusters allow multiple hosts to access the same data on the shared storage. Because most operating systems lack support for simultaneous data access by multiple hosts, parallel clusters are usually accomplished by use of special versions of software and special releases of applications. For example, Oracle Parallel Server is a version of Oracle's database that has been designed to run on a parallel cluster. Each machine runs Oracle, and a layer of software tracks access to the shared disk. Each machine has full access to all data in the database. To provide this shared access to data, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a **distributed lock manager (DLM)**, is included in some cluster technology.

Cluster technology is changing rapidly. Some cluster products support dozens of systems in a cluster, as well as clustered nodes that are separated by miles. Many of these improvements are made possible by **storage-area networks (SANs)**, as described in Section 12.3.3, which allow many systems to attach to a pool of storage. If the applications and their data are stored on the SAN, then the cluster software can assign the application to run on any host that is attached to the SAN. If the host fails, then any other host can take over. In a database cluster, dozens of hosts can share the same database, greatly increasing performance and reliability.

## 1.4 Operating-System Structure

Now that we have discussed basic information about computer-system organization and architecture, we are ready to talk about operating systems. An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. There are, however, many commonalities, which we consider in this section.

One of the most important aspects of operating systems is the ability to multiprogram. A single user cannot, in general, keep either the CPU or the I/O devices busy at all times. **Multiprogramming** increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.

The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.7). This set of jobs can be a subset of the jobs kept in the job pool—which contains all jobs that enter the system—since the number of jobs that can be kept simultaneously in memory is usually smaller than the number of jobs that can be kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a

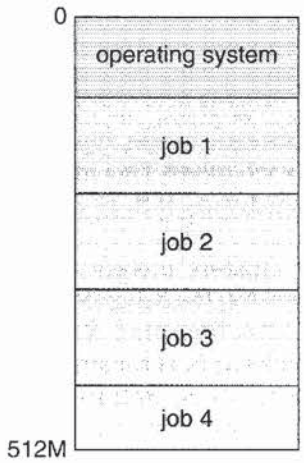



**Figure 1.7** Memory layout for a multiprogramming system.

non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to, and executes, another job. When *that* job needs to wait, the CPU is switched to *another* job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If he has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.)

Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system. **Time sharing** (or **multitasking**) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an **interactive** (or **hands-on**) **computer system**, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard or a mouse, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into

memory and executing is called a **process**. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or other device. Since interactive I/O typically runs at "people speeds," it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.

Time-sharing and multiprogramming require several jobs to be kept simultaneously in memory. Since in general main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision is **job scheduling,** which is discussed in Chapter 5. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management, which is covered in Chapters 8 and 9. In addition, if several jobs are ready to run at the same time, the system must choose among them. Making this decision is **CPU scheduling,** which is discussed in Chapter 5. Finally, running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. These considerations are discussed throughout the text.

In a time-sharing system, the operating system must ensure reasonable response time, which is sometimes accomplished through **swapping**, where processes are swapped in and out of main memory to the disk. A more common method for achieving this goal is **virtual memory**, a technique that allows the execution of a process that is not completely in memory (Chapter 9). The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

Time-sharing systems must also provide a file system (Chapters 10 and 11). The file system resides on a collection of disks; hence, disk management must be provided (Chapter 12). Also, time-sharing systems provide a mechanism for protecting resources from inappropriate use (Chapter 14). To ensure orderly execution, the system must provide mechanisms for job synchronization and communication (Chapter 6), and it may ensure that jobs do not get stuck in a deadlock, forever waiting for one another (Chapter 7).

## 1.5 Operating-System Operations

As mentioned earlier, modern operating systems are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt

or a trap. A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt.

Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program that was running. With sharing, many processes could be adversely affected by a bug in one program. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes. More subtle errors can occur in a multiprogramming system, where one erroneous program might modify another program, the data of another program, or even the operating system itself.

Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

### 1.5.1 Dual-Mode Operation

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user-defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution.

At the very least, we need two separate **modes** of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), it must transition from user to kernel mode to fulfill the request. This is shown in Figure 1.8. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that

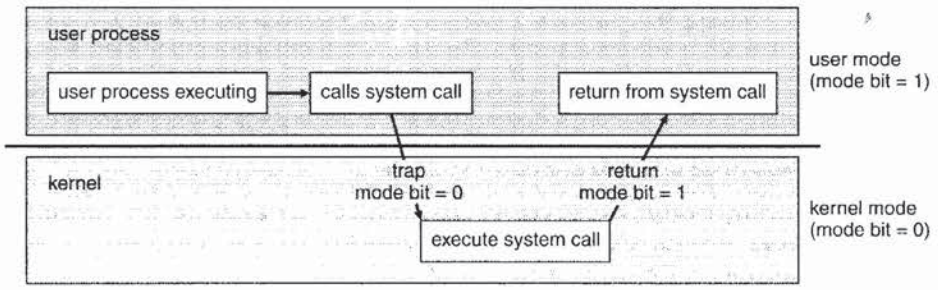



**Figure 1.8** Transition from user to kernel mode.

may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The instruction to switch to user mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management. As we shall see throughout the text, there are many additional privileged instructions.

We can now see the life cycle of instruction execution in a computer system. Initial control is within the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic `trap` instruction, although some systems (such as the MIPS R2000 family) have a specific `syscall` instruction.

When a system call is executed, it is treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call. We describe system calls more fully in Section 2.3.

The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system. For instance, MS-DOS was written for the Intel 8088 architecture, which has no mode bit and therefore no dual mode. A user program running awry can wipe out the operating system by writing over it with data; and multiple programs are able to write to a device at the same time,

with possibly disastrous results. Recent versions of the Intel CPU, such as the Pentium, do provide dual-mode operation. Accordingly, most contemporary operating systems, such as Microsoft Windows 2000 and Windows XP, and Linux and Solaris for x86 systems, take advantage of this feature and provide greater protection for the operating system.

Once hardware protection is in place, errors violating modes are detected by the hardware. These errors are normally handled by the operating system. If a user program fails in some way—such as by making an attempt either to execute an illegal instruction or to access memory that is not in the user's address space—then the hardware will trap to the operating system. The trap transfers control through the interrupt vector to the operating system, just as an interrupt does. When a program error occurs, the operating system must terminate the program abnormally. This situation is handled by the same code as is a user-requested abnormal termination. An appropriate error message is given, and the memory of the program may be dumped. The memory dump is usually written to a file so that the user or programmer can examine it and perhaps correct it and restart the program.

### 1.5.2 Timer

We must ensure that the operating system maintains control over the CPU. We must prevent a user program from getting stuck in an infinite loop or not calling system services and never returning control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

Thus, we can use the timer to prevent a user program from running too long. A simple technique is to initialize a counter with the amount of time that a program is allowed to run. A program with a 7-minute time limit, for example, would have its counter initialized to 420. Every second, the timer interrupts and the counter is decremented by 1. As long as the counter is positive, control is returned to the user program. When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.

## 1.6 Process Management

A program does nothing unless its instructions are executed by a CPU. A program in execution, as mentioned, is a process. A time-shared user program such as a compiler is a process. A word-processing program being run by an

individual user on a PC is a process. A system task, such as sending output to a printer, can also be a process (or at least part of one). For now, you can consider a process to be a job or a time-shared program, but later you will learn that the concept is more general. As we shall see in Chapter 3, it is possible to provide system calls that allow processes to create subprocesses to execute concurrently.

A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are either given to the process when it is created or allocated to it while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along. For example, consider a process whose function is to display the status of a file on the screen of a terminal. The process will be given as an input the name of the file and will execute the appropriate instructions and system calls to obtain and display on the terminal the desired information. When the process terminates, the operating system will reclaim any reusable resources.

We emphasize that a program by itself is not a process; a program is a *passive* entity, such as the contents of a file stored on disk, whereas a process is an *active* entity. A single-threaded process has one **program counter** specifying the next instruction to execute. (Threads will be covered in Chapter 4.) The execution of such a process must be sequential. The CPU executes one instruction of the process after another, until the process completes. Further, at any time, one instruction at most is executed on behalf of the process. Thus, although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

A process is the unit of work in a system. Such a system consists of a collection of 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). All these processes can potentially execute concurrently—by multiplexing the CPU among them on a single CPU, for example.

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

We discuss process-management techniques in Chapters 3 through 6.

## 1.7 Memory Management

As we discussed in Section 1.2.2, the main memory is central to the operation of a modern computer system. Main memory is a large array of words or bytes,

ranging in size from hundreds of thousands to billions. Each word or byte has its own address. Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on a Von Neumann architecture). The main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from disk, those data must first be transferred to main memory by CPU-generated I/O calls. In the same way, instructions must be in memory for the CPU to execute them.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed.

To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management. Many different memory-management schemes are used. These schemes reflect various approaches, and the effectiveness of any given algorithm depends on the situation. In selecting a memory-management scheme for a specific system, we must take into account many factors—especially on the *hardware* design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating and deallocating memory space as needed

Memory-management techniques will be discussed in Chapters 8 and 9.

## 1.8 Storage Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **file**. The operating system maps files onto physical media and accesses these files via the storage devices.

### 1.8.1 File-System Management

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Magnetic disk, optical disk, and magnetic tape are the most common. Each of these media has its own characteristics and physical organization. Each medium is controlled by a device, such as a disk drive or tape drive, that

also has its own unique characteristics. These properties include access speed, capacity, data-transfer rate, and access method (sequential or random).

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form (for example, text files), or they may be formatted rigidly (for example, fixed fields). Clearly, the concept of a file is an extremely general one.

The operating system implements the abstract concept of a file by managing mass storage media, such as tapes and disks, and the devices that control them. Also, files are normally organized into directories to make them easier to use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways (for example, read, write, append) files may be accessed.

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media

File-management techniques will be discussed in Chapters 10 and 11.

### 1.8.2 Mass-Storage Management

As we have already seen, because main memory is too small to accommodate all data and programs, and because the data that it holds are lost when power is lost, the computer system must provide secondary storage to back up main memory. Most modern computer systems use disks as the principal on-line storage medium for both programs and data. Most programs—including compilers, assemblers, word processors, editors, and formatters—are stored on a disk until loaded into memory and then use the disk as both the source and destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system. The operating system is responsible for the following activities in connection with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

Because secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may hinge on the speeds of the disk subsystem and of the algorithms that manipulate that subsystem.

There are, however, many uses for storage that is slower and lower in cost (and sometimes of higher capacity) than secondary storage. Backups of disk data, seldom-used data, and long-term archival storage are some examples.

Magnetic tape drives and their tapes and CD and DVD drives and platters are typical **tertiary storage** devices. The media (tapes and optical platters) vary between **WORM** (write-once, read-many-times) and **RW** (read–write) formats.

Tertiary storage is not crucial to system performance, but it still must be managed. Some operating systems take on this task, while others leave tertiary-storage management to application programs. Some of the functions that operating systems can provide include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

Techniques for secondary and tertiary storage management will be discussed in Chapter 12.

### 1.8.3 Caching

**Caching** is an important principle of computer systems. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—on a temporary basis. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache; if it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

In addition, internal programmable registers, such as index registers, provide a high-speed cache for main memory. The programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory. There are also caches that are implemented totally in hardware. For instance, most systems have an instruction cache to hold the next instructions expected to be executed. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory. For similar reasons, most systems have one or more high-speed data caches in the memory hierarchy. We are not concerned with these hardware-only caches in this text, since they are outside the control of the operating system.

Because caches have limited size, **cache management** is an important design problem. Careful selection of the cache size and of a replacement policy can result in greatly increased performance. See Figure 1.9 for a storage performance comparison in large workstations and small servers that shows the need for caching. Various replacement algorithms for software-controlled caches are discussed in Chapter 9.

Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use, and data must be in main memory before being moved to secondary storage for safekeeping. The file-system data, which resides permanently on secondary storage, may appear on several levels in the storage hierarchy. At the highest level, the operating system may maintain a cache of file-system data in main memory. Also, electronic RAM disks (also known as **solid-state disks**) may be used for high-speed storage that is accessed through the file-system interface. The bulk of secondary storage is on magnetic disks. The magnetic-disk storage, in turn, is often backed up onto magnetic tapes or removable disks to protect against data loss in case of a hard-disk failure. Some systems automatically

| Level | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Name | registers | cache | main memory | disk storage |
| Typical size | < 1 KB | > 16 MB | > 16 GB | > 100 GB |
| Implementation technology | custom memory with multiple ports, CMOS | on-chip or off-chip CMOS SRAM | CMOS DRAM | magnetic disk |
| Access time (ns) | 0.25 – 0.5 | 0.5 – 25 | 80 – 250 | 5,000.000 |
| Bandwidth (MB/sec) | 20,000 – 100,000 | 5000 – 10,000 | 1000 – 5000 | 20 – 150 |
| Managed by | compiler | hardware | operating system | operating system |
| Backed by | cache | main memory | disk | CD or tape |

**Figure 1.9** Performance of various levels of storage.

archive old file data from secondary storage to tertiary storage, such as tape jukeboxes, to lower the storage cost (see Chapter 12).

The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. In contrast, transfer of data from disk to memory is usually controlled by the operating system.

In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on magnetic disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register (see Figure 1.10). Once the increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk.

In a computing environment where only one process executes at a time, this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy. However, in a multitasking environment, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.

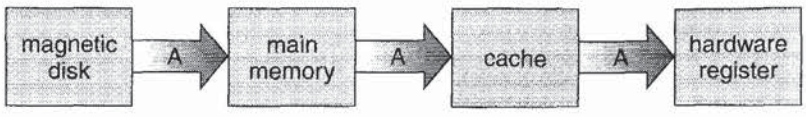



**Figure 1.10** Migration of integer A from disk to register.

The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache. In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute concurrently, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency**, and it is usually a hardware problem (handled below the operating-system level).

In a distributed environment, the situation becomes even more complex. In this environment, several copies (or replicas) of the same file can be kept on different computers that are distributed in space. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible. There are various ways to achieve this guarantee, as we discuss in Chapter 17.

### 1.8.4 I/O Systems

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the **I/O subsystem**. The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

We discussed in Section 1.2.3 how interrupt handlers and device drivers are used in the construction of efficient I/O subsystems. In Chapter 13, we discuss how the I/O subsystem interfaces to the other system components, manages devices, transfers data, and detects I/O completion.

## 1.9 Protection and Security

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system. For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU without eventually relinquishing control. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

**Protection**, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must

provide means for specification of the controls to be imposed and means for enforcement.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage, as we discuss in Chapter 14.

A system can have adequate protection but still be prone to failure and allow inappropriate access. Consider a user whose authentication information (her means of identifying herself to the system) is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of-service attacks (which use all of a system's resources and so keep legitimate users out of the system), identity theft, and theft of service (unauthorized use of a system). Prevention of some of these attacks is consider an operating-system function on some systems, while others leave the prevention to policy or additional software. Due to the alarming rise in security incidents, operating-system security features represent a fast-growing area of research and of implementation. Security is discussed in Chapter 15.

Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifiers (user IDs)**. In Windows NT parlance, this is a **security ID (SID)**. These numerical IDs are unique, one per user. When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of the user's processes and threads. When an ID needs to be user readable, it is translated back to the user name via the user name list.

In some circumstances, we wish to distinguish among sets of users rather than individual users. For example, the owner of a file on a UNIX system may be allowed to issue all operations on that file, whereas a selected set of users may only be allowed to read the file. To accomplish this, we need to define a group name and the set of users belonging to that group. Group functionality can be implemented as a system-wide list of group names and **group identifiers**. A user can be in one or more groups, depending on operating-system design decisions. The user's group IDs are also included in every associated process and thread.

In the course of normal use of a system, the user ID and group ID for a user are sufficient. However, a user sometimes needs to **escalate privileges** to gain extra permissions for an activity. The user may need access to a device that is restricted, for example. Operating systems provide various methods to allow privilege escalation. On UNIX, for example, the `setuid` attribute on a program causes that program to run with the user ID of the owner of the file, rather than the current user's ID. The process runs with this **effective UID** until it turns off the extra privileges or terminates. Consider an example of how this is done in Solaris 10. User pbg has user ID 101 and group ID 14, which are assigned via /etc/passwd:

```
pbg:x:101:14::/export/home/pbg:/usr/bin/bash
```

## 1.10 Distributed Systems

A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide the users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability. Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface's device driver. Others make users specifically invoke network functions. Generally, systems contain a mix of the two modes—for example FTP and NFS. The protocols that create a distributed system can greatly affect that system's utility and popularity.

A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. TCP/IP is the most common network protocol, although ATM and other protocols are in widespread use. Likewise, operating-system support of protocols varies. Most operating systems support TCP/IP, including the Windows and UNIX operating systems. Some systems support proprietary protocols to suit their needs. To an operating system, a network protocol simply needs an interface device—a network adapter, for example—with a device driver to manage it, as well as software to handle data. These concepts are discussed throughout this book.

Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a floor, or a building. A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide. These networks may run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks. For example, a **metropolitan-area network (MAN)** could link buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **small-area network** such as might be found in a home.

The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network. Even very short-range infrared communication can be used for networking. At a rudimentary level, whenever computers communicate, they use or create a network. These networks also vary in their performance and reliability.

Some operating systems have taken the concept of networks and distributed systems further than the notion of providing network connectivity. A **network operating system** is an operating system that provides features such as file sharing across the network and that includes a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers. A distributed operating system provides a less autonomous environment: The different operating

systems communicate closely enough to provide the illusion that only a single operating system controls the network.

We cover computer networks and distributed systems in Chapters 16 through 18.

## 1.11 Special-Purpose Systems

The discussion thus far has focused on general-purpose computer systems that we are all familiar with. There are, however, different classes of computer systems whose functions are more limited and whose objective is to deal with limited computation domains.

### 1.11.1 Real-Time Embedded Systems

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to VCRs and microwave ovens. They tend to have very specific tasks. The systems they run on are usually primitive, and so the operating systems provide limited features. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

These embedded systems vary considerably. Some are general-purpose computers, running standard operating systems—such as UNIX—with special-purpose applications to implement the functionality. Others are hardware devices with a special-purpose embedded operating system providing just the functionality desired. Yet others are hardware devices with application-specific integrated circuits (**ASICs**) that perform their tasks without an operating system.

The use of embedded systems continues to expand. The power of these devices, both as standalone units and as members of networks and the Web, is sure to increase as well. Even now, entire houses can be computerized, so that a central computer—either a general-purpose computer or an embedded system—can control heating and lighting, alarm systems, and even coffee makers. Web access can enable a home owner to tell the house to heat up before she arrives home. Someday, the refrigerator may call the grocery store when it notices the milk is gone.

Embedded systems almost always run **real-time operating systems**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing *must* be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt *after* it had smashed into the car it was building. A real-time system functions correctly only if it

returns the correct result within its time constraints. Contrast this system with a time-sharing system, where it is desirable (but not mandatory) to respond quickly, or a batch system, which may have no time constraints at all.

In Chapter 19, we cover real-time embedded systems in great detail. In Chapter 5, we consider the scheduling facility needed to implement real-time functionality in an operating system. In Chapter 9, we describe the design of memory management for real-time computing. Finally, in Chapter 22, we describe the real-time components of the Windows XP operating system.

### 1.11.2 Multimedia Systems

Most operating systems are designed to handle conventional data such as text files, programs, word-processing documents, and spreadsheets. However, a recent trend in technology is the incorporation of **multimedia data** into computer systems. Multimedia data consist of audio and video files as well as conventional files. These data differ from conventional data in that multimedia data—such as frames of video—must be delivered (streamed) according to certain time restrictions (for example, 30 frames per second).

Multimedia describes a wide range of applications that are in popular use today. These include audio files such as MP3 DVD movies, video conferencing, and short video clips of movie previews or news stories downloaded over the Internet. Multimedia applications may also include live webcasts (broadcasting over the World Wide Web) of speeches or sporting events and even live webcams that allow a viewer in Manhattan to observe customers at a cafe in Paris. Multimedia applications need not be either audio or video; rather, a multimedia application often includes a combination of both. For example, a movie may consist of separate audio and video tracks. Nor must multimedia applications be delivered only to desktop personal computers. Increasingly, they are being directed toward smaller devices, including PDAs and cellular telephones. For example, a stock trader may have stock quotes delivered wirelessly and in real time to his PDA.

In Chapter 20, we explore the demands of multimedia applications, how multimedia data differ from conventional data, and how the nature of these data affects the design of operating systems that support the requirements of multimedia systems.

### 1.11.3 Handheld Systems

**Handheld systems** include personal digital assistants (PDAs), such as Palm and Pocket-PCs, and cellular telephones, many of which use special-purpose embedded operating systems. Developers of handheld systems and applications face many challenges, most of which are due to the limited size of such devices. For example, a PDA is typically about 5 inches in height and 3 inches in width, and it weighs less than one-half pound. Because of their size, most handheld devices have a small amount of memory, slow processors, and small display screens. We will take a look now at each of these limitations.

The amount of physical memory in a handheld depends upon the device, but typically is is somewhere between 512 KB and 128 MB. (Contrast this with a typical PC or workstation, which may have several gigabytes of memory!) As a result, the operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory

manager when the memory is not being used. In Chapter 9, we will explore virtual memory, which allows developers to write programs that behave as if the system has more memory than is physically available. Currently, not many handheld devices use virtual memory techniques, so program developers must work within the confines of limited physical memory.

A second issue of concern to developers of handheld devices is the speed of the processor used in the devices. Processors for most handheld devices run at a fraction of the speed of a processor in a PC. Faster processors require more power. To include a faster processor in a handheld device would require a larger battery, which would take up more space and would have to be replaced (or recharged) more frequently. Most handheld devices use smaller, slower processors that consume less power. Therefore, the operating system and applications must be designed not to tax the processor.

The last issue confronting program designers for handheld devices is I/O. A lack of physical space limits input methods to small keyboards, handwriting recognition, or small screen-based keyboards. The small display screens limit output options. Whereas a monitor for a home computer may measure up to 30 inches, the display for a handheld device is often no more than 3 inches square. Familiar tasks, such as reading e-mail and browsing web pages, must be condensed into smaller displays. One approach for displaying the content in web pages is **web clipping**, where only a small subset of a web page is delivered and displayed on the handheld device.

Some handheld devices use wireless technology, such as BlueTooth or 802.11, allowing remote access to e-mail and web browsing. Cellular telephones with connectivity to the Internet fall into this category. However, for PDAs that do not provide wireless access, downloading data typically requires the user to first download the data to a PC or workstation and then download the data to the PDA. Some PDAs allow data to be directly copied from one device to another using an infrared link.

Generally, the limitations in the functionality of PDAs are balanced by their convenience and portability. Their use continues to expand as network connections become more available and other options, such as digital cameras and MP3 players, expand their utility.

## 1.12 Computing Environments

So far, we have provided an overview of computer-system organization and major operating-system components. We conclude with a brief overview of how these are used in a variety of computing environments.

### 1.12.1 Traditional Computing

As computing matures, the lines separating many of the traditional computing environments are blurring. Consider the "typical office environment." Just a few years ago, this environment consisted of PCs connected to a network, with servers providing file and print services. Remote access was awkward, and portability was achieved by use of laptop computers. Terminals attached to mainframes were prevalent at many companies as well, with even fewer remote access and portability options.

The current trend is toward providing more ways to access these computing environments. Web technologies are stretching the boundaries of traditional computing. Companies establish **portals**, which provide web accessibility to their internal servers. **Network computers** are essentially terminals that understand web-based computing. Handheld computers can synchronize with PCs to allow very portable use of company information. Handheld PDAs can also connect to **wireless networks** to use the company's web portal (as well as the myriad other web resources).

At home, most users had a single computer with a slow modem connection to the office, the Internet, or both. Today, network-connection speeds once available only at great cost are relatively inexpensive, giving home users more access to more data. These fast data connections are allowing home computers to serve up web pages and to run networks that include printers, client PCs, and servers. Some homes even have **firewalls** to protect their networks from security breaches. Those firewalls cost thousands of dollars a few years ago and did not even exist a decade ago.

In the latter half of the previous century, computing resources were scarce. (Before that, they were nonexistent!) For a period of time, systems were either batch or interactive. Batch system processed jobs in bulk, with predetermined input (from files or other sources of data). Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. Time-sharing systems used a timer and scheduling algorithms to rapidly cycle processes through the CPU, giving each user a share of the resources.

Today, traditional time-sharing systems are uncommon. The same scheduling technique is still in use on workstations and servers, but frequently the processes are all owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time. Consider the windows created while a user is working on a PC, for example, and the fact that they may be performing different tasks at the same time.

### 1.12.2 Client–Server Computing

As PCs have become faster, more powerful, and cheaper, designers have shifted away from centralized system architecture. Terminals connected to centralized systems are now being supplanted by PCs. Correspondingly, user-interface functionality once handled directly by the centralized systems is increasingly being handled by the PCs. As a result, many of todays systems act as **server systems** to satisfy requests generated by **client systems**. This form of specialized distributed system, called **client–server** system, has the general structure depicted in Figure 1.11.

Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data); in response, the server executes the action and sends back results to the client. A server running a database that responds to client requests for data is an example of such a system.

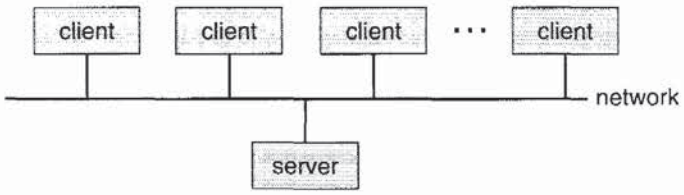



**Figure 1.11** General structure of a client–server system.

- The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers.

### 1.12.3 Peer-to-Peer Computing

Another structure for a distributed system is the peer-to-peer (P2P) system model. In this model, clients and servers are not distinguished from one another; instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client-server systems. In a client-server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network. Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- A peer acting as a client must first discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a *discovery protocol* must be provided that allows peers to discover services provided by other peers in the network.

Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella, that enable peers to exchange files with one another. The Napster system uses an approach similar to the first type described above: a centralized server maintains an index of all files stored on peer nodes in the Napster network, and the actual exchanging of files takes place between the peer nodes. The Gnutella system uses a technique similar to the second type: a client broadcasts file requests to other nodes in the system, and nodes that can service the request respond directly to the client. The future of exchanging files remains uncertain because

many of the files are copyrighted (music, for example), and there are laws governing the distribution of copyrighted material. In any case, though, peer-to-peer technology undoubtedly will play a role in the future of many services, such as searching, file exchange, and e-mail.

### 1.12.4 Web-Based Computing

The Web has become ubiquitous, leading to more access by a wider variety of devices than was dreamt of a few years ago. PCs are still the most prevalent access devices, with workstations, handheld PDAs, and even cell phones also providing access.

Web computing has increased the emphasis on networking. Devices that were not previously networked now include wired or wireless access. Devices that were networked now have faster network connectivity, provided by either improved networking technology, optimized network implementation code, or both.

The implementation of web-based computing has given rise to new categories of devices, such as **load balancers**, which distribute network connections among a pool of similar servers. Operating systems like Windows 95, which acted as web clients, have evolved into Linux and Windows XP, which can act as web servers as well as clients. Generally, the Web has increased the complexity of devices, because their users require them to be web-enabled.

## 1.13 Summary

An operating system is software that manages the computer hardware as well as providing an environment for application programs to run. Perhaps the most visible aspect of an operating system is the interface to the computer system it provides to the human user.

For a computer to do its job of executing programs, the programs must be in main memory. Main memory is the only large storage area that the processor can access directly. It is an array of words or bytes, ranging in size from millions to billions. Each word in memory has its own address. The main memory is usually a volatile storage device that loses its contents when power is turned off or lost. Most computer systems provide secondary storage as an extension of main memory. Secondary storage provides a form of non-volatile storage that is capable of holding large quantities of data permanently. The most common secondary-storage device is a magnetic disk, which provides storage of both programs and data.

The wide variety of storage systems in a computer system can be organized in a hierarchy according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.

There are several different strategies for designing a computer system. Uniprocessor systems have only a single processor while multiprocessor systems contain two or more processors that share physical memory and peripheral devices. The most common multiprocessor design is symmetric multiprocessing (or SMP), where all processors are considered peers and run

independently of one another. Clustered systems are a specialized form of multiprocessor systems and consist of multiple computer systems connected by a local area network.

To best utilize the CPU, modern operating systems employ multiprogramming, which allows several jobs to be in memory at the same time, thus ensuring the CPU always has a job to execute. Timesharing systems are an extension of multiprogramming whereby CPU scheduling algorithms rapidly switch between jobs, thus providing the illusion each job is running concurrently.

The operating system must ensure correct operation of the computer system. To prevent user programs from interfering with the proper operation of the system, the hardware has two modes: user mode and kernel mode. Various instructions (such as I/O instructions and halt instructions) are privileged and can be executed only in kernel mode. The memory in which the operating system resides must also be protected from modification by the user. A timer prevents infinite loops. These facilities (dual mode, privileged instructions, memory protection, and timer interrupt) are basic building blocks used by operating systems to achieve correct operation.

A process (or job) is the fundamental unit of work in an operating system. Process management includes creating and deleting processes and providing mechanisms for processes to communicate and synchronize with another. An operating system manages memory by keeping track of what parts of memory are being used and by whom. The operating system is also responsible for dynamically allocating and freeing memory space. Storage space is also managed by the operating system and this includes providing file systems for representing files and directories and managing space on mass storage devices.

Operating systems must also be concerned with protecting and securing the operating system and users. Protection are mechanisms that control the access of processes or users to the resources made available by the computer system. Security measures are responsible for defending a computer system from external or internal attacks.

Distributed systems allow users to share resources on geographically dispersed hosts connected via a computer network. Services may be provided through either the client–server model or the peer-to-peer model. In a clustered system, multiple machines can perform computations on data residing on shared storage, and computing can continue even when some subset of cluster members fails.

LANs and WANs are the two basic types of networks. LANs enable processors distributed over a small geographical area to communicate, whereas WANs allow processors distributed over a larger area to communicate. LANs typically are faster than WANs.

There are several computer systems that serve specific purposes. These include real-time operating systems designed for embedded environments such as consumer devices, automobiles, and robotics. Real-time operating systems have well defined, fixed time constraints. Processing *must* be done within the defined constraints, or the system will fail. Multimedia systems involve the delivery of multimedia data and often have special requirements of displaying or playing audio, video, or synchronized audio and video streams.

Recently, the influence of the Internet and the World Wide Web has encouraged the development of modern operating systems that include web browsers and networking and communication software as integral features.

## Exercises

**1.1** In a multiprogramming and time-sharing environment, several users share the system simultaneously. This situation can result in various security problems.

a. What are two such problems?

b. Can we ensure the same degree of security in a time-shared machine as in a dedicated machine? Explain your answer.

**1.2** The issue of resource utilization shows up in different forms in different types of operating systems. List what resources must be managed carefully in the following settings:

a. Mainframe or minicomputer systems

b. Workstations connected to servers

c. Handheld computers

**1.3** Under what circumstances would a user be better off using a time-sharing system rather than a PC or single-user workstation?

**1.4** Which of the functionalities listed below need to be supported by the operating system for the following two settings: (a) handheld devices and (b) real-time systems.

a. Batch programming

b. Virtual memory

c. Time sharing

**1.5** Describe the differences between symmetric and asymmetric multiprocessing. What are three advantages and one disadvantage of multiprocessor systems?

**1.6** How do clustered systems differ from multiprocessor systems? What is required for two machines belonging to a cluster to cooperate to provide a highly available service?

**1.7** Distinguish between the client–server and peer-to-peer models of distributed systems.

**1.8** Consider a computing cluster consisting of two nodes running a database. Describe two ways in which the cluster software can manage access to the data on the disk. Discuss the benefits and disadvantages of each.

**1.9** How are network computers different from traditional personal computers? Describe some usage scenarios in which it is advantageous to use network computers.

**1.10** What is the purpose of interrupts? What are the differences between a trap and an interrupt? Can traps be generated intentionally by a user program? If so, for what purpose?

**1.11** Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.

a. How does the CPU interface with the device to coordinate the transfer?

b. How does the CPU know when the memory operations are complete?

c. The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

**1.12** Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computer systems? Give arguments both that it is and that it is not possible.

**1.13** Give two reasons why caches are useful. What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?

**1.14** Discuss, with examples, how the problem of maintaining coherence of cached data manifests itself in the following processing environments:

a. Single-processor systems

b. Multiprocessor systems

c. Distributed systems

**1.15** Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

**1.16** What network configuration would best suit the following environments?

a. A dormitory floor

b. A university campus

c. A state

d. A nation

**1.17** Define the essential properties of the following types of operating systems:

a. Batch

b. Interactive

c. Time sharing

d. Real time

e. Network

f. Parallel

g. Distributed

h. Clustered

i. Handheld

**1.18** What are the tradeoffs inherent in handheld computers?

## Bibliographical Notes

Brookshear [2003] provides an overview of computer science in general.

An overview of the Linux operating system is presented in Bovet and Cesati [2002]. Solomon and Russinovich [2000] give an overview of Microsoft Windows and considerable technical detail about the system internals and components. Mauro and McDougall [2001] cover the Solaris operating system. Mac OS X is presented at http://www.apple.com/macosx.

Coverage of peer-to-peer systems includes Parameswaran et al. [2001], Gong [2002], Ripeanu et al. [2002], Agre [2003], Balakrishnan et al. [2003], and Loo [2003]. A discussion on peer-to-peer file-sharing systems can be found in Lee [2003]. A good coverage of cluster computing is presented by Buyya [1999]. Recent advances in cluster computing are described by Ahmed [2000]. A survey of issues relating to operating systems support for distributed systems can be found in Tanenbaum and Van Renesse [1985].

Many general textbooks cover operating systems, including Stallings [2000b], Nutt [2004] and Tanenbaum [2001].

Hamacher et al. [2002] describes computer organization. Hennessy and Patterson [2002] provide coverage of I/O systems and buses, and of system architecture in general.

Cache memories, including associative memory, are described and analyzed by Smith [1982]. That paper also includes an extensive bibliography on the subject.

Discussions concerning magnetic-disk technology are presented by Freedman [1983] and by Harker et al. [1981]. Optical disks are covered by Kenville [1982], Fujitani [1984], O'Leary and Kitts [1985], Gait [1988], and Olsen and Kenley [1989]. Discussions of floppy disks are offered by Pechura and Schoeffler [1983] and by Sarisky [1983]. General discussions concerning mass-storage technology are offered by Chi [1982] and by Hoagland [1985].

Kurose and Ross [2005], Tanenbaum [2003], Peterson and Davie [1996], and Halsall [1992] provide general overviews of computer networks. Fortier [1989] presents a detailed discussion of networking hardware and software.

Wolf [2003] discusses recent developments in developing embedded systems. Issues related to handheld devices can be found in Myers and Beigl [2003] and Di Pietro and Mancini [2003].

# Operating-System Structures




An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

We can view an operating system from several vantage points. One view focuses on the services that the system provides; another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections. In this chapter, we explore all three aspects of operating systems, showing the viewpoints of users, programmers, and operating-system designers. We consider what services an operating system provides, how they are provided, and what the various methodologies are for designing such systems. Finally, we describe how operating systems are created and how a computer starts its operating system.

## CHAPTER OBJECTIVES

- To describe the services an operating system provides to users, processes, and other systems.
- To discuss the various ways of structuring an operating system.
- To explain how operating systems are installed and customized and how they boot.

## 2.1 Operating-System Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided, of course, differ from one operating system to another, but we can identify common classes. These operating-system services are provided for the convenience of the programmer, to make the programming task easier.

One set of operating-system services provides functions that are helpful to the user.

- **User interface**. Almost all operating systems have a **user interface (UI)**. This interface can take several forms. One is a **command-line interface (CLI)**, which uses text commands and a method for entering them (say, a program to allow entering and editing of commands). Another is a **batch interface**, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a **graphical user interface (GUI)** is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Some systems provide two or all three of these variations.
- **Program execution**. The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
- **I/O operations**. A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a CRT screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.
- **File-system manipulation**. The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some programs include permissions management to allow or deny access to files or directories based on file ownership.
- **Communications**. There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via *shared memory* or through *message passing*, in which packets of information are moved between processes by the operating system.
- **Error detection**. The operating system needs to be constantly aware of possible errors. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Debugging facilities can greatly enhance the user's and programmer's abilities to use the system efficiently.

Another set of operating-system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

- **Resource allocation**. When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors. There may also be routines to allocate printers, modems, USB storage drives, and other peripheral devices.
- **Accounting**. We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.
- **Protection and security**. The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including modems and network adapters, from invalid access attempts and to recording all such connections for detection of break-ins. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

## 2.2 User Operating-System Interface

There are two fundamental approaches for users to interface with the operating system. One technique is to provide a command-line interface or **command interpreter** that allows users to directly enter commands that are to be performed by the operating system. The second approach allows the user to interface with the operating system via a graphical user interface or GUI.

### 2.2.1 Command Interpreter

Some operating systems include the command interpreter in the kernel. Others, such as Windows XP and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as **shells**. For example, on UNIX and Linux systems, there are several different shells a user may choose from including the *Bourne shell*, *C shell*, *Bourne-Again shell*, the *Korn shell*, etc. Most shells provide similar functionality with only minor differences; most users choose a shell based upon personal preference.

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The MS-DOS and UNIX shells operate in this way. There are two general ways in which these commands can be implemented.

In one approach, the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.

An alternative approach—used by UNIX, among other operating systems—implements most commands through system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. Thus, the UNIX command to delete a file

```
rm file.txt
```

would search for a file called `rm`, load the file into memory, and execute it with the parameter `file.txt`. The function associated with the `rm` command would be defined completely by the code in the file `rm`. In this way, programmers can add new commands to the system easily by creating new files with the proper names. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

### 2.2.2 Graphical User Interfaces

A second strategy for interfacing with the operating system is through a user-friendly graphical user interface or GUI. Rather than having users directly enter commands via a command-line interface, a GUI allows provides a mouse-based window-and-menu system as an interface. A GUI provides a **desktop** metaphor where the mouse is moved to position its pointer on images, or **icons**, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a **folder**—or pull down a menu that contains commands.

Graphical user interfaces first appeared due in part to research taking place in the early 1970s at Xerox PARC research facility. The first GUI appeared on the Xerox Alto computer in 1973. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s. The user interface to the Macintosh operating system (Mac OS) has undergone various changes over the years, the most significant being the adoption of the *Aqua* interface that appeared with Mac OS X. Microsoft's first version of Windows—version 1.0—was based upon a GUI interface to the MS-DOS operating system. The various versions of Windows systems proceeding this initial version have made cosmetic changes to the appearance of the GUI and several enhancements to its functionality, including the Windows Explorer.

Traditionally, UNIX systems have been dominated by command-line interfaces, although there are various GUI interfaces available, including the Common Desktop Environment (CDE) and X-Windows systems that are common on

commercial versions of UNIX such as Solaris and IBM's AIX system. However, there has been significant development in GUI designs from various **open-source** projects such as *K Desktop Environment* (or *KDE*) and the *GNOME* desktop by the GNU project. Both the KDE and GNOME desktops run on Linux and various UNIX systems and are available under open-source licenses, which means their source code is in the public domain.

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. As a very general rule, many UNIX users prefer a command-line interface as they often provide powerful shell interfaces. Alternatively, most Windows users are pleased to use the Windows GUI environment and almost never use the MS-DOS shell interface. The various changes undergone by the Macintosh operating systems provides a nice study in contrast. Historically, Mac OS has not provided a command line interface, always requiring its users to interface with the operating system using its GUI. However, with the release of Mac OS X (which is in part implemented using a UNIX kernel), the operating system now provides both a new Aqua interface and command-line interface as well.

The user interface can vary from system to system and even from user to user within a system. It typically is substantially removed from the actual system structure. The design of a useful and friendly user interface is therefore not a direct function of the operating system. In this book, we concentrate on the fundamental problems of providing adequate service to user programs. From the point of view of the operating system, we do not distinguish between user programs and system programs.

## 2.3 System Calls

**System calls** provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly), may need to be written using assembly-language instructions.

Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is for the program to ask the user for the names of the two files. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls.

Once the two file names are obtained, the program must open the input file and create the output file. Each of these operations requires another system call. There are also possible error conditions for each operation. When the program tries to open the input file, it may find that there is no file of that name or that

the file is protected against access. In these cases, the program should print a message on the console (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.

Now that both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors, depending on the output device (no more disk space, printer out of paper, and so on).

Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call). As we can see, even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. This system-call sequence is shown in Figure 2.1.

Most programmers never see this level of detail, however. Typically, application developers design programs according to an **application programming interface (API)**. The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each

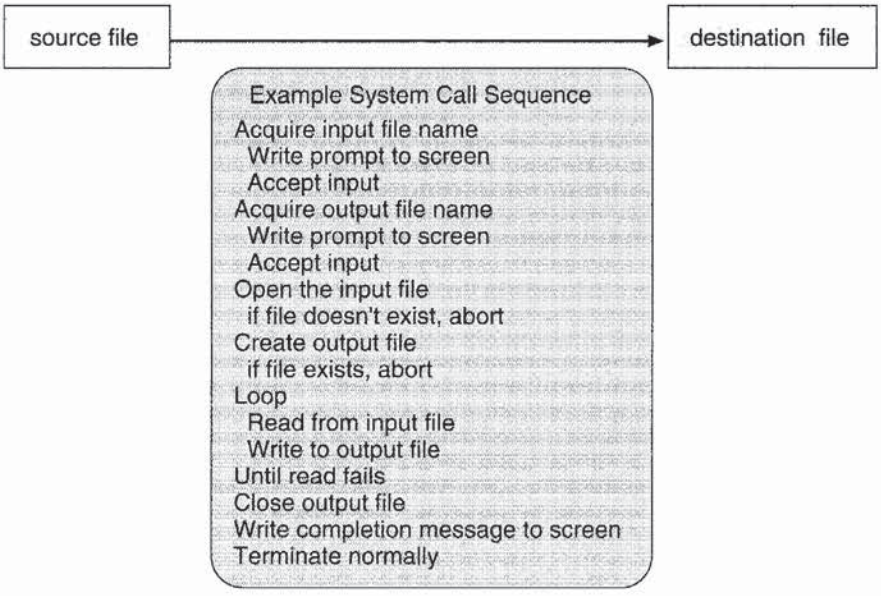



**Figure 2.1** Example of how system calls are used.

### EXAMPLE OF STANDARD API

As an example of a standard API, consider the ReadFile() function in the Win32 API—a function for reading from a file. The API for this function appears in Figure 2.2.

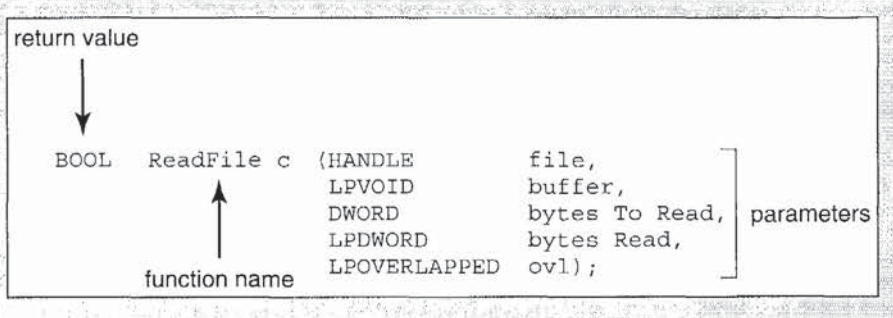



**Figure 2.2** The API for the ReadFile() function.

A description of the parameters passed to ReadFile() is as follows:

- HANDLE file—the file to be read.
- LPVOID buffer—a buffer where the data will be read into and written from.
- DWORD bytesToRead—the number of bytes to be read into the buffer.
- LPDWORD bytesRead—the number of bytes read during the last read.
- LPOVERLAPPED ovl—indicates if overlapped I/O is being used.

function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Win32 API for Windows systems, the POSIX API for POSIX-based systems (which includes virtually all versions of UNIX, Linux, and Mac OS X), and the Java API for designing programs that run on the Java virtual machine.

Note that the system-call names used throughout this text are generic examples. Each operating system has its own name for each system call.

Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. For example, the Win32 function CreateProcess() (which unsurprisingly is used to create a new process) actually calls the NTCreateProcess() system call in the Windows kernel. Why would an application programmer prefer programming according to an API rather than invoking actual system calls? There are several reasons for doing so. One benefit of programming according to an API concerns program portability: An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API (although in reality, architectural differences often make this more difficult than it may appear). Furthermore, actual system calls can often be more detailed

and difficult to work with than the API available to an application programmer. Regardless, there often exists a strong correlation between invoking a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Win32 APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

The run-time support system (a set of functions built into libraries included with a compiler) for most programming languages provides a **system-call interface** that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system call within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system call interface then invokes the intended system call in the operating system kernel and returns the status of the system call and any return values.

The caller needs to know nothing about how the system call is implemented or what it does during execution. Rather, it just needs to obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library. The relationship between an API, the system-call interface, and the operating system is shown in Figure 2.3, which illustrates how the operating system handles a user application invoking the open() system call.

System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call. For example, to get input, we may need to specify the file or device to use as the source, as well as the address and

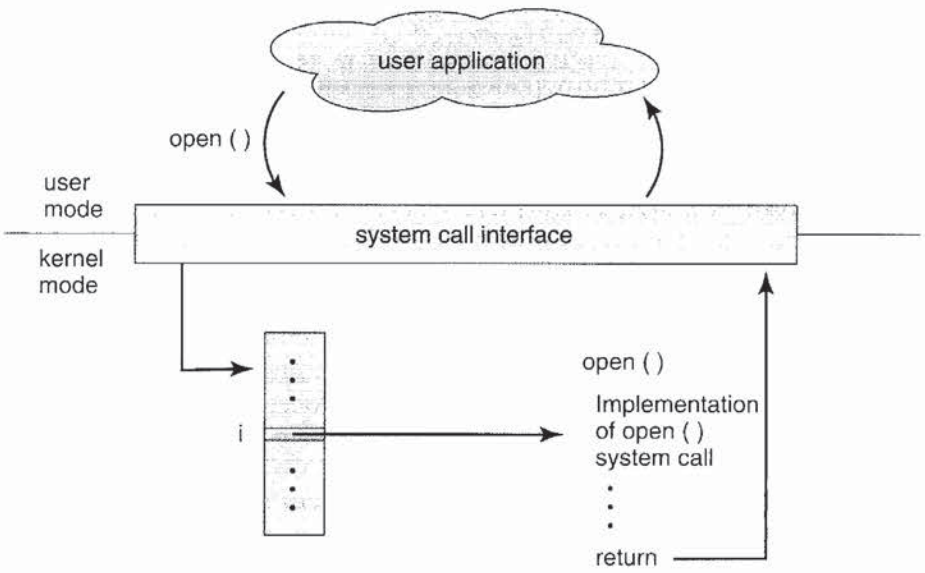



**Figure 2.3** The handling of a user application invoking the open() system call.

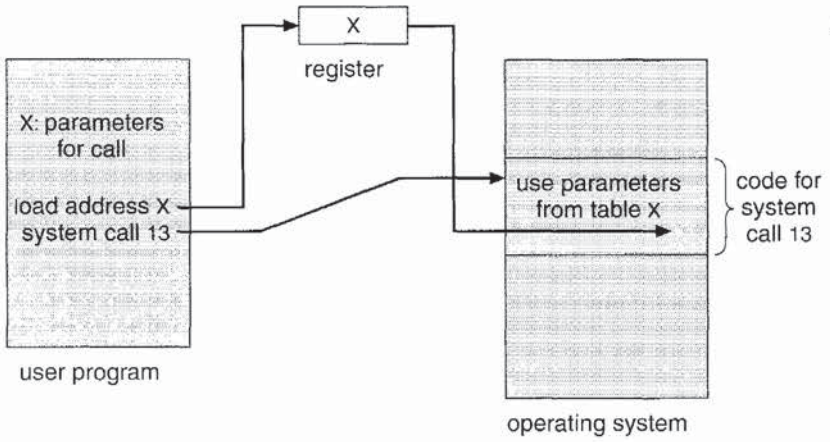



**Figure 2.4** Passing of parameters as a table.

length of the memory buffer into which the input should be read. Of course, the device or file and length may be implicit in the call.

Three general methods are used to pass parameters to the operating system. The simplest approach is to pass the parameters in *registers*. In some cases, however, there may be more parameters than registers. In these cases, the parameters are generally stored in a *block*, or table, in memory, and the address of the block is passed as a parameter in a register (Figure 2.4). This is the approach taken by Linux and Solaris. Parameters also can be placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system. Some operating systems prefer the block or stack method, because those approaches do not limit the number or length of parameters being passed.

## 2.4 Types of System Calls

System calls can be grouped roughly into five major categories: **process control**, **file manipulation**, **device manipulation**, **information maintenance**, and **communications**. In Sections 2.4.1 through 2.4.5, we discuss briefly the types of system calls that may be provided by an operating system. Most of these system calls support, or are supported by, concepts and functions that are discussed in later chapters. Figure 2.5 summarizes the types of system calls normally provided by an operating system.

### 2.4.1 Process Control

A running program needs to be able to halt its execution either normally (`end`) or abnormally (`abort`). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a **debugger**—a system program designed to aid the programmer in finding and correcting bugs—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

**Figure 2.5** Types of system calls.

invoking command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error. In a GUI system, a pop-up window might alert the user to the error and ask for guidance. In a batch system, the command interpreter usually terminates the entire job and continues with the next job.

### EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call(s) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program. This is shown in Figure 2.6.

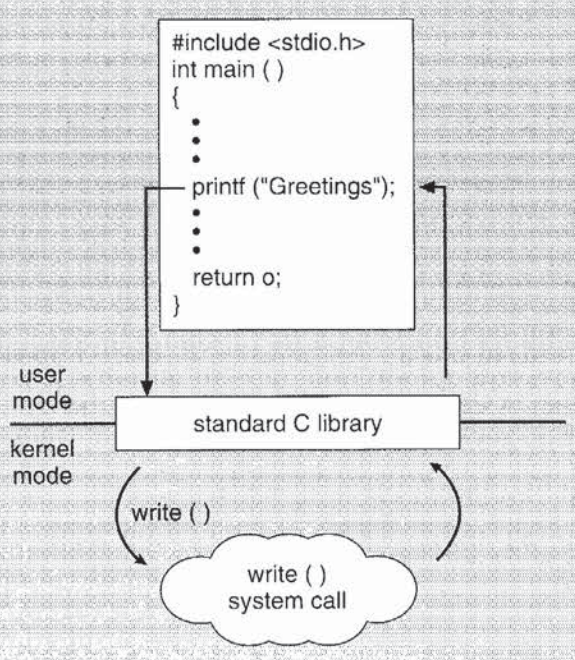



**Figure 2.6** C library handling of `write()`.

Some systems allow control cards to indicate special recovery actions in case an error occurs. A **control card** is a batch system concept. It is a command to manage the execution of a process. If the program discovers an error in its input and wants to terminate abnormally, it may also want to define an error level. More severe errors can be indicated by a higher-level error parameter. It is then possible to combine normal and abnormal termination by defining a normal termination as an error at level 0. The command interpreter or a following program can use this error level to determine the next action automatically.

A process or job executing one program may want to `load` and `execute` another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command, the click of a mouse, or a batch command. An interesting question is where to return control when the loaded program terminates. This question is related to the problem of whether the existing program is lost, saved, or allowed to continue execution concurrently with the new program.

If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new job or process to be multiprogrammed. Often, there is a system call specifically for this purpose (`create process` or `submit job`).

If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (`get process attributes` and `set process attributes`). We may also want to terminate a job or process that we created (`terminate process`) if we find that it is incorrect or is no longer needed.

Having created new jobs or processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (`wait time`); more probably, we will want to wait for a specific event to occur (`wait event`). The jobs or processes should then signal when that event has occurred (`signal event`). System calls of this type, dealing with the coordination of concurrent processes, are discussed in great detail in Chapter 6.

Another set of system calls is helpful in debugging a program. Many systems provide system calls to `dump` memory. This provision is useful for debugging. A program `trace` lists each instruction as it is executed; it is provided by fewer systems. Even microprocessors provide a CPU mode known as *single step*, in which a trap is executed by the CPU after every instruction. The trap is usually caught by a debugger.

Many operating systems provide a time profile of a program to indicate the amount of time that the program executes at a particular location or set of locations. A time profile requires either a tracing facility or regular timer interrupts. At every occurrence of the timer interrupt, the value of the program

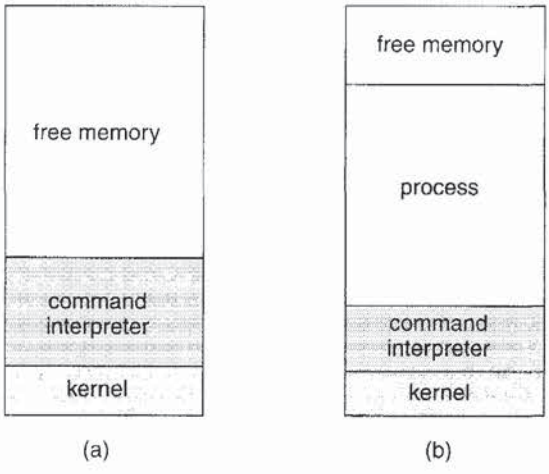



**Figure 2.7** MS-DOS execution. (a) At system startup. (b) Running a program.

counter is recorded. With sufficiently frequent timer interrupts, a statistical picture of the time spent on various parts of the program can be obtained.

There are so many facets of and variations in process and job control that we next use two examples—one involving a single-tasking system and the other a multitasking system—to clarify these concepts. The MS-DOS operating system is an example of a single-tasking system. It has a command interpreter that is invoked when the computer is started (Figure 2.7(a)). Because MS-DOS is single-tasking, it uses a simple method to run a program and does not create a new process. It loads the program into memory, writing over most of itself to give the program as much memory as possible (Figure 2.7(b)). Next, it sets the instruction pointer to the first instruction of the program. The program then runs, and either an error causes a trap, or the program executes a system call to terminate. In either case, the error code is saved in the system memory for later use. Following this action, the small portion of the command interpreter that was not overwritten resumes execution. Its first task is to reload the rest of the command interpreter from disk. Then the command interpreter makes the previous error code available to the user or to the next program.

FreeBSD (derived from Berkeley UNIX) is an example of a multitasking system. When a user logs on to the system, the shell of the user's choice is run. This shell is similar to the MS-DOS shell in that it accepts commands and executes programs that the user requests. However, since FreeBSD is a multitasking system, the command interpreter may continue running while another program is executed (Figure 2.8). To start a new process, the shell executes a `fork()` system call. Then, the selected program is loaded into memory via an `exec()` system call, and the program is executed. Depending on the way the command was issued, the shell then either waits for the process to finish or runs the process "in the background." In the latter case, the shell immediately requests another command. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files or through a GUI interface. Meanwhile, the user is free to ask the shell to run other programs, to monitor the progress of the running process, to change that program's priority,

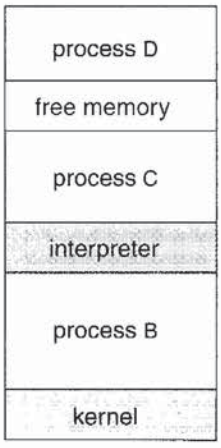



**Figure 2.8** FreeBSD running multiple programs.

### SOLARIS 10 DYNAMIC TRACING FACILITY

Making running operating systems easier to understand, debug, and tune is an active area of operating system research and implementation. For example, Solaris 10 includes the `dtrace` dynamic tracing facility. This facility dynamically adds probes to a running system. These probes can be queried via the D programming language to determine an astonishing amount about the kernel, the system state, and process activities. For example, Figure 2.9 follows an application as it executes a system call (`ioctl`) and further shows the functional calls within the kernel as they execute to perform the system call. Lines ending with "U" are executed in user mode, and lines ending in "K" in kernel mode.

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                        U
  0   -> _XEventsQueued                     U
  0     -> _X11TransBytesReadable           U
  0     <- _X11TransBytesReadable           U
  0     -> _X11TransSocketBytesReadable     U
  0     <- _X11TransSocketBytesreadable     U
  0     -> ioctl                            U
  0       -> ioctl                          K
  0         -> getf                         K
  0           -> set_active_fd              K
  0           <- set_active_fd              K
  0         <- getf                         K
  0         -> get_udatamodel               K
  0         <- get_udatamodel               K
...
  0         -> releasef                     K
  0           -> clear_active_fd            K
  0           <- clear_active_fd            K
  0           -> cv_broadcast               K
  0           <- cv_broadcast               K
  0         <- releasef                     K
  0       <- ioctl                          K
  0     <- ioctl                            U
  0   <- _XEventsQueued                     U
  0 <- XEventsQueued                        U
```

**Figure 2.9** Solaris 10 dtrace follows a system call within the kernel.

Other operating systems are starting to include various performance and tracing tools, fostered by research at various institutions, including the Paradyn project.

and so on. When the process is done, it executes an `exit()` system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code. This status or error code is then available to the shell or other programs. Processes are discussed in Chapter 3 with an program example using the `fork()` and `exec()` system calls.

### 2.4.2 File Management

The file system will be discussed in more detail in Chapters 10 and 11. We can, however, identify several common system calls dealing with files,

We first need to be able to `create` and `delete` files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to `open` it and to use it. We may also `read`, `write`, or `reposition` (rewinding or skipping to the end of the file, for example). Finally, we need to `close` the file, indicating that we are no longer using it.

We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on. At least two system calls, `get file attribute` and `set file attribute`, are required for this function. Some operating systems provide many more calls, such as calls for file `move` and `copy`. Others might provide an API that performs those operations using code and other system calls, and others might just provide system programs to perform those tasks. If the system programs are callable by other programs, then each can be considered an API by other system programs.

### 2.4.3 Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.

The various resources controlled by the operating sysstem can be thought of as devices. Some of these devices are physical devices (for example, tapes), while others can be thought of as abstract or virtual devices (for example, files). If there are multiple users of the system, the system may require us to first `request` the device, to ensure exclusive use of it. After we are finished with the device, we `release` it. These functions are similar to the `open` and `close` system calls for files. Other operating systems allow unmanaged access to devices. The hazard then is the potential for device contention and perhaps deadlock, which is described in Chapter 7.

Once the device has been requested (and allocated to us), we can `read`, `write`, and (possibly) `reposition` the device, just as we can with files. In fact, the similarity between I/O devices and files is so great that many operating systems, including UNIX, merge the two into a combined file–device structure. In this case, a set of system calls is used on files and devices. Sometimes, I/O devices are identified by special file names, directory placement, or file attributes.

The UI can also make files and devices appear to be similar, even though the underlying system calls are dissimilar. This is another example of the many design decisions that go into building an operating system and user interface.

### 2.4.4 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most

systems have a system call to return the current `time` and `date`. Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

In addition, the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (`get process attributes` and `set process attributes`). In Section 3.1.3, we discuss what information is normally kept.

### 2.4.5 Communication

There are two common models of interprocess communication: the message-passing model and the shared-memory model. In the **message-passing model**, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a *host name* by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a *process name*, and this name is translated into an identifier by which the operating system can refer to the process. The `get hostid` and `get processid` system calls do this translation. The identifiers are then passed to the general-purpose `open` and `close` calls provided by the file system or to specific `open connection` and `close connection` system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an `accept connection` call. Most processes that will be receiving connections are special-purpose *daemons*, which are systems programs provided for that purpose. They execute a `wait for connection` call and are awakened when a connection is made. The source of the communication, known as the *client*, and the receiving daemon, known as a *server*, then exchange messages by using `read message` and `write message` system calls. The `close connection` call terminates the communication.

In the **shared-memory model**, processes use `shared memory create` and `shared memory attach` system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by the processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously. Such mechanisms are discussed in Chapter 6. In Chapter 4, we look at a variation of the process scheme—threads—in which memory is shared by default.

Both of the models just discussed are common in operating systems, and most systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared

memory allows maximum speed and convenience of communication, since it can be done at memory speeds when it takes place within a computer. Problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

## 2.5 System Programs

Another aspect of a modern system is the collection of system programs. Recall Figure 1.1, which depicted the logical computer hierarchy. At the lowest level is hardware. Next is the operating system, then the system programs, and finally the application programs. System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex. They can be divided into these categories:

- **File management**. These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information**. Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a **registry**, which is used to store and retrieve configuration information.
- **File modification**. Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.
- **Programming-language support**. Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.
- **Program loading and execution**. Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.
- **Communications**. These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

In addition to systems programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and

statistical-analysis packages, and games. These programs are known as **system utilities** or **application programs**.

The view of the operating system seen by most users is defined by the application and system programs, rather than by the actual system calls. Consider PCs. When his computer is running the Mac OS X operating system, a user might see the GUI, featuring a mouse and windows interface. Alternatively, or even in one of the windows, he might have a command-line UNIX shell. Both use the same set of system calls, but the system calls look different and act in different ways.

## 2.6 Operating-System Design and Implementation

In this section, we discuss problems we face in designing and implementing an operating system. There are, of course, no complete solutions to such problems, but there are approaches that have proved successful.

### 2.6.1 Design Goals

The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: batch, time shared, single user, multiuser, distributed, real time, or general purpose.

Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups: *user* goals and *system* goals.

Users desire certain obvious properties in a system: The system should be convenient to use, easy to learn and to use, reliable, safe, and fast. Of course, these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them.

A similar set of requirements can be defined by those people who must design, create, maintain, and operate the system: The system should be easy to design, implement, and maintain; it should be flexible, reliable, error free, and efficient. Again, these requirements are vague and may be interpreted in various ways.

There is, in short, no unique solution to the problem of defining the requirements for an operating system. The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments. For example, the requirements for VxWorks, a real-time operating system for embedded systems, must have been substantially different from those for MVS, a large multiuser, multiaccess operating system for IBM mainframes.

Specifying and designing an operating system is a highly creative task. Although no textbook can tell you how to do it, general principles have been developed in the field of **software engineering**, and we turn now to a discussion of some of these principles.

### 2.6.2 Mechanisms and Policies

One important principle is the separation of **policy** from **mechanism**. Mechanisms determine *how* to do something; policies determine *what* will be done.

For example, the timer construct (see Section 1.5.2) is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

The separation of policy and mechanism is important for flexibility. Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism insensitive to changes in policy would be more desirable. A change in policy would then require redefinition of only certain parameters of the system. For instance, consider a mechanism for giving priority to certain types of programs over others. If the mechanism is properly separated from policy, it can be used to support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones or to support the opposite policy.

Microkernel-based operating systems (Section 2.7.3) take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building blocks. These blocks are almost policy free, allowing more advanced mechanisms and policies to be added via user-created kernel modules or via user programs themselves. As an example, consider the history of UNIX. At first, it had a time-sharing scheduler. In the latest version of Solaris, scheduling is controlled by loadable tables. Depending on the table currently loaded, the system can be time shared, batch processing, real time, fair share, or any combination. Making the scheduling mechanism general purpose allows vast policy changes to be made with a single `load-new-table` command. At the other extreme is a system such as Windows, in which both mechanism and policy are encoded in the system to enforce a global look and feel. All applications have similar interfaces, because the interface itself is built into the kernel and system libraries. The Mac OS X operating system has similar functionality.

Policy decisions are important for all resource allocation. Whenever it is necessary to decide whether or not to allocate a resource, a policy decision must be made. Whenever the question is *how* rather than *what*, it is a mechanism that must be determined.

### 2.6.3 Implementation

Once an operating system is designed, it must be implemented. Traditionally, operating systems have been written in assembly language. Now, however, they are most commonly written in higher-level languages such as C or C++.

The first system that was not written in assembly language was probably the Master Control Program (MCP) for Burroughs computers. MCP was written in a variant of ALGOL. MULTICS, developed at MIT, was written mainly in PL/1. The Linux and Windows XP operating systems are written mostly in C, although there are some small sections of assembly code for device drivers and for saving and restoring the state of registers.

The advantages of using a higher-level language, or at least a systems-implementation language, for implementing operating systems are the same as those accrued when the language is used for application programs: The code can be written faster, is more compact, and is easier to understand and debug. In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an operating system is far easier to *port*—to move to some other hardware—

if it is written in a higher-level language. For example, MS-DOS was written in Intel 8088 assembly language. Consequently, it is available on only the Intel family of CPUs. The Linux operating system, in contrast, is written mostly in C and is available on a number of different CPUs, including Intel 80X86, Motorola 680X0, SPARC, and MIPS RX000.

The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements. This, however, is no longer a major issue in today's systems. Although an expert assembly-language programmer can produce efficient small routines, for large programs a modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code. Modern processors have deep pipelining and multiple functional units that can handle complex dependencies that can overwhelm the limited ability of the human mind to keep track of details.

As is true in other systems, major performance improvements in operating systems are more likely to be the result of better data structures and algorithms than of excellent assembly-language code. In addition, although operating systems are large, only a small amount of the code is critical to high performance; the memory manager and the CPU scheduler are probably the most critical routines. After the system is written and is working correctly, bottleneck routines can be identified and can be replaced with assembly-language equivalents.

To identify bottlenecks, we must be able to monitor system performance. Code must be added to compute and display measures of system behavior. In a number of systems, the operating system does this task by producing trace listings of system behavior. All interesting events are logged with their time and important parameters and are written to a file. Later, an analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies. These same traces can be run as input for a simulation of a suggested improved system. Traces also can help people to find errors in operating-system behavior.

## 2.7 Operating-System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components rather than have one monolithic system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions. We have already discussed briefly in Chapter 1 the common components of operating systems. In this section, we discuss how these components are interconnected and melded into a kernel.

### 2.7.1 Simple Structure

Many commercial systems do not have well-defined structures. Frequently, such operating systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in

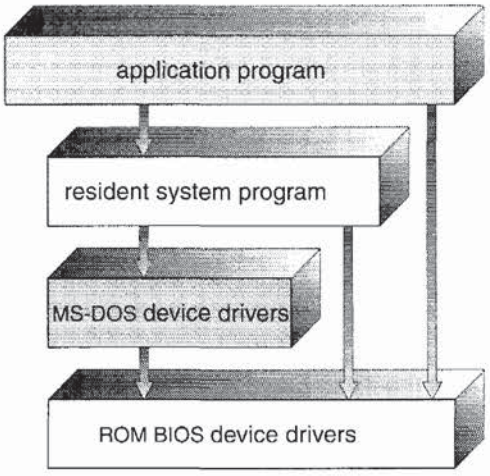



**Figure 2.10** MS-DOS layer structure.

the least space, so it was not divided into modules carefully. Figure 2.10 shows its structure.

In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

Another example of limited structuring is the original UNIX operating system. UNIX is another system that initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered, as shown in Figure 2.11. Everything below the system call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one level. This monolithic structure was difficult to implement and maintain.

### 2.7.2 Layered Approach

With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS or UNIX systems. The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems. Under the top-down approach, the overall functionality and features are determined and are

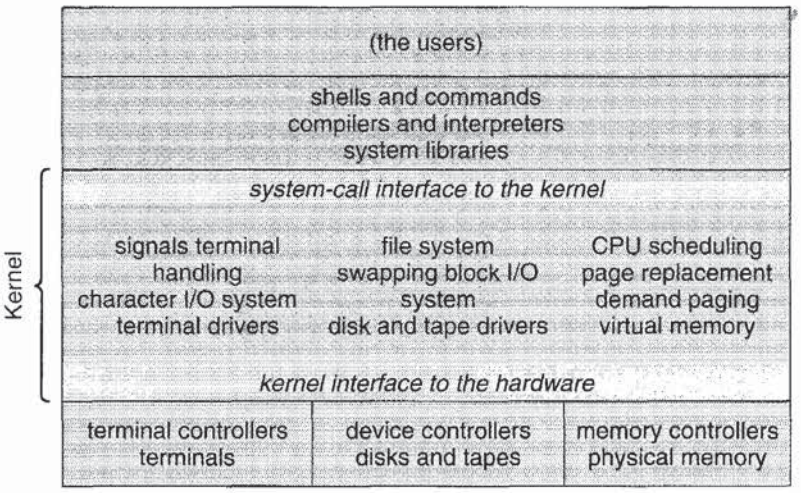



**Figure 2.11** UNIX system structure.

separated into components. Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken up into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer $N$) is the user interface. This layering structure is depicted in Figure 2.12.

An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer $M$—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer $M$, in turn, can invoke operations on lower-level layers.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified.

Each layer is implemented with only those operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary. For example, the device driver for the backing

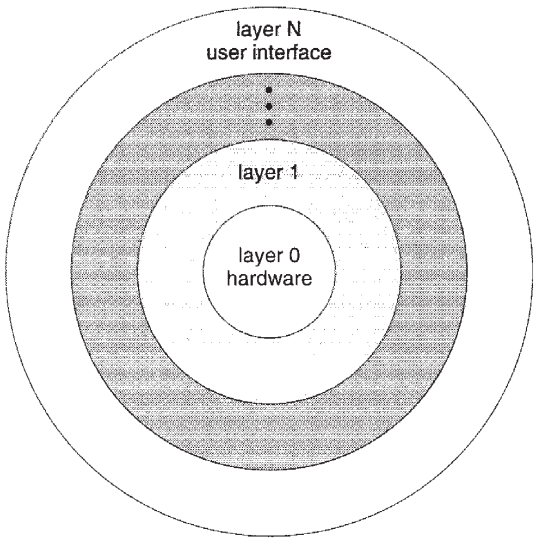



**Figure 2.12** A layered operating system.

store (disk space used by virtual-memory algorithms) must be at a lower level than the memory-management routines, because memory management requires the ability to use the backing store.

Other requirements may not be so obvious. The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. However, on a large system, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of memory, requiring the backing-store driver routine to be below the CPU scheduler.

A final problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware. At each layer, the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call; the net result is a system call that takes longer than does one on a nonlayered system.

These limitations have caused a small backlash against layering in recent years. Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the difficult problems of layer definition and interaction.

### 2.7.3 Microkernels

We have already seen that as UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach. This method structures the operating system by removing all nonessential components from the kernel and

implementing them as system and user-level programs. The result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, microkernels provide minimal process and memory management, in addition to a communication facility.

The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space. Communication is provided by *message passing*, which was described in Section 2.4.5. For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

One benefit of the microkernel approach is ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

Several contemporary operating systems have used the microkernel approach. Tru64 UNIX (formerly Digital UNIX) provides a UNIX interface to the user, but it is implemented with a Mach kernel. The Mach kernel maps UNIX system calls into messages to the appropriate user-level services.

Another example is QNX. QNX is a real-time operating system that is also based on the microkernel design. The QNX microkernel provides services for message passing and process scheduling. It also handles low-level network communication and hardware interrupts. All other services in QNX are provided by standard processes that run outside the kernel in user mode.

Unfortunately, microkernels can suffer from performance decreases due to increased system function overhead. Consider the history of Windows NT. The first release had a layered microkernel organization. However, this version delivered low performance compared with that of Windows 95. Windows NT 4.0 partially redressed the performance problem by moving layers from user space to kernel space and integrating them more closely. By the time Windows XP was designed, its architecture was more monolithic than microkernel.

### 2.7.4 Modules

Perhaps the best current methodology for operating-system design involves using object-oriented programming techniques to create a modular kernel. Here, the kernel has a set of core components and dynamically links in additional services either during boot time or during run time. Such a strategy uses dynamically loadable modules and is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X. For example, the Solaris operating system structure, shown in Figure 2.13, is organized around a core kernel with seven types of loadable kernel modules:

1. Scheduling classes
2. File systems

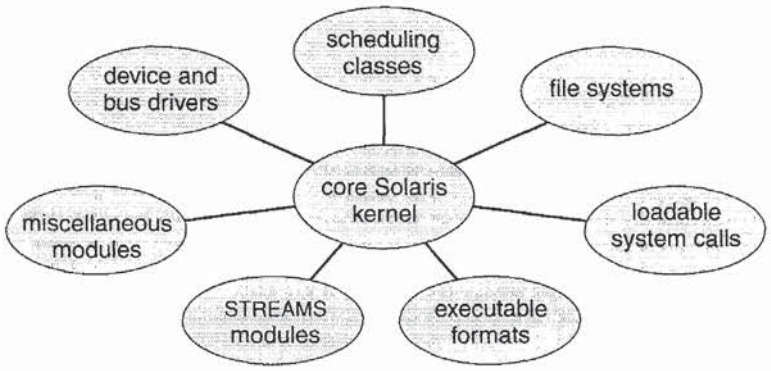



**Figure 2.13** Solaris loadable modules.

3. Loadable system calls
4. Executable formats
5. STREAMS modules
6. Miscellaneous
7. Device and bus drivers

Such a design allows the kernel to provide core services yet also allows certain features to be implemented dynamically. For example, device and bus drivers for specific hardware can be added to the kernel, and support for different file systems can be added as loadable modules. The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system in that any module can call any other module. Furthermore, the approach is like the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

The Apple Macintosh Mac OS X operating system uses a hybrid structure. Mac OS X (also known as *Darwin*) structures the operating system using a layered technique where one layer consists of the Mach microkernel. The structure of Mac OS X appears in Figure 2.14.

The top layers include application environments and a set of services providing a graphical interface to applications. Below these layers is the kernel environment, which consists primarily of the Mach microkernel and the BSD kernel. Mach provides memory management; support for remote procedure calls (RPCs) and interprocess communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads. In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which Mac OS X refers to as **kernel extensions**). As shown in the figure, applications and common services can make use of either the Mach or BSD facilities directly.

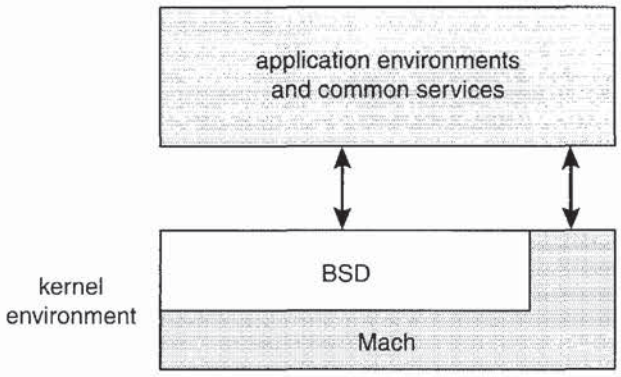



**Figure 2.14** The Mac OS X structure.

## 2.8 Virtual Machines

The layered approach described in Section 2.7.2 is taken to its logical conclusion in the concept of a **virtual machine**. The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.

By using CPU scheduling (Chapter 5) and virtual-memory techniques (Chapter 9), an operating system can create the illusion that a process has its own processor with its own (virtual) memory. Normally, a process has additional features, such as system calls and a file system, that are not provided by the bare hardware. The virtual-machine approach does not provide any such additional functionality but rather provides an interface that is *identical* to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer (Figure 2.15).

There are several reasons for creating a virtual machine, all of which are fundamentally related to being able to share the same hardware yet run several different execution environments (that is, different operating systems) concurrently. We will explore the advantages of virtual machines in more detail in Section 2.8.2. Throughout much of this section, we discuss the VM operating system for IBM systems, as it provides a useful working example; furthermore IBM pioneered the work in this area.

A major difficulty with the virtual-machine approach involves disk systems. Suppose that the physical machine has three disk drives but wants to support seven virtual machines. Clearly, it cannot allocate a disk drive to each virtual machine, because the virtual-machine software itself will need substantial disk space to provide virtual memory and spooling. The solution is to provide virtual disks—termed *minidisks* in IBM's VM operating system—that are identical in all respects except size. The system implements each minidisk by allocating as many tracks on the physical disks as the minidisk needs. Obviously, the sum of the sizes of all minidisks must be smaller than the size of the physical disk space available.

Users thus are given their own virtual machines. They can then run any of the operating systems or software packages that are available on the underlying

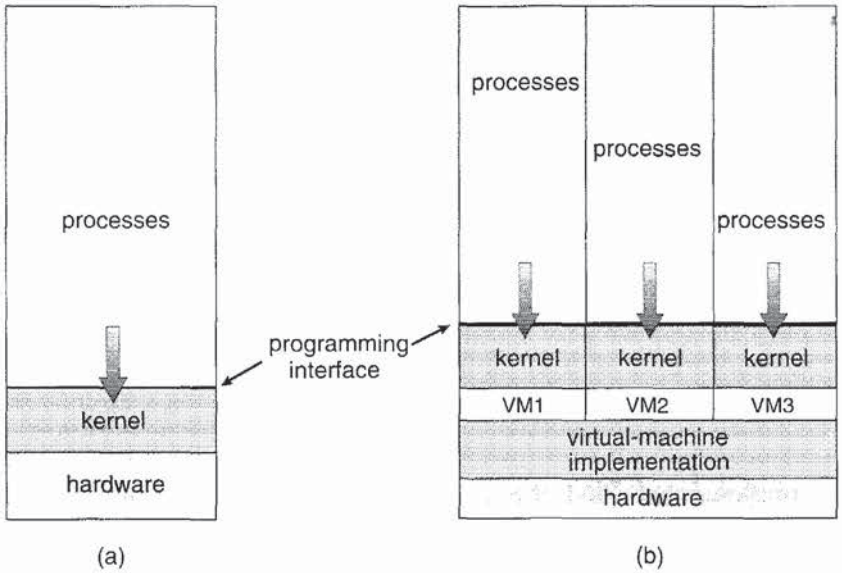



**Figure 2.15** System models. (a) Nonvirtual machine. (b) Virtual machine.

machine. For the IBM VM system, a user normally runs CMS—a single-user interactive operating system. The virtual-machine software is concerned with multiprogramming multiple virtual machines onto a physical machine, but it does not need to consider any user-support software. This arrangement may provide a useful way to divide the problem of designing a multiuser interactive system into two smaller pieces.

### 2.8.1 Implementation

Although the virtual-machine concept is useful, it is difficult to implement. Much work is required to provide an *exact* duplicate of the underlying machine. Remember that the underlying machine has two modes: user mode and kernel mode. The virtual-machine software can run in kernel mode, since it is the operating system. The virtual machine itself can execute in only user mode. Just as the physical machine has two modes, however, so must the virtual machine. Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in a physical user mode. Those actions that cause a transfer from user mode to kernel mode on a real machine (such as a system call or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual kernel mode on a virtual machine.

Such a transfer can be accomplished as follows. When a system call, for example, is made by a program running on a virtual machine in virtual user mode, it will cause a transfer to the virtual-machine monitor in the real machine. When the virtual-machine monitor gains control, it can change the register contents and program counter for the virtual machine to simulate the effect of the system call. It can then restart the virtual machine, noting that it is now in virtual kernel mode.

The major difference, of course, is time. Whereas the real I/O might have taken 100 milliseconds, the virtual I/O might take less time (because it is

spooled) or more time (because it is interpreted). In addition, the CPU is being multiprogrammed among many virtual machines, further slowing down the virtual machines in unpredictable ways. In the extreme case, it may be necessary to simulate all instructions to provide a true virtual machine. VM works for IBM machines because normal instructions for the virtual machines can execute directly on the hardware. Only the privileged instructions (needed mainly for I/O) must be simulated and hence execute more slowly.

### 2.8.2 Benefits

The virtual-machine concept has several advantages. Notice that, in this environment, there is complete protection of the various system resources. Each virtual machine is completely isolated from all other virtual machines, so there are no protection problems. At the same time, however, there is no direct sharing of resources. Two approaches to provide sharing have been implemented. First, it is possible to share a minidisk and thus to share files. This scheme is modeled after a physical shared disk but is implemented by software. Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network. Again, the network is modeled after physical communication networks but is implemented in software.

Such a virtual-machine system is a perfect vehicle for operating-systems research and development. Normally, changing an operating system is a difficult task. Operating systems are large and complex programs, and it is difficult to be sure that a change in one part will not cause obscure bugs in some other part. The power of the operating system makes changing it particularly dangerous. Because the operating system executes in kernel mode, a wrong change in a pointer could cause an error that would destroy the entire file system. Thus, it is necessary to test all changes to the operating system carefully.

The operating system, however, runs on and controls the entire machine. Therefore, the current system must be stopped and taken out of use while changes are made and tested. This period is commonly called *system-development time*. Since it makes the system unavailable to users, system-development time is often scheduled late at night or on weekends, when system load is low.

A virtual-machine system can eliminate much of this problem. System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on a physical machine. Normal system operation seldom needs to be disrupted for system development.

### 2.8.3 Examples

Despite the advantages of virtual machines, they received little attention for a number of years after they were first developed. Today, however, virtual machines are coming back into fashion as a means of solving system compatibility problems. In this section, we explore two popular contemporary virtual machines: VMware and the Java virtual machine. As we will see, these virtual machines typically run on top of an operating system of any of the design types discussed earlier. Thus, operating system design methods—

simple layers, microkernel, modules, and virtual machines—are not mutually exclusive.

#### 2.8.3.1 VMware

VMware is a popular commercial application that abstracts Intel 80X86 hardware into isolated virtual machines. VMware runs as an application on a host operating system such as Windows or Linux and allows this host system to concurrently run several different **guest operating systems** as independent virtual machines.

Consider the following scenario: A developer has designed an application and would like to test it on Linux, FreeBSD, Windows NT, and Windows XP. One option is for her to obtain four different computers, each running a copy of one of these operating systems. Another alternative is for her first to install Linux on a computer system and test the application, then to install FreeBSD and test the application, and so forth. This option allows her to use the same physical computer but is time-consuming, since she must install a new operating system for each test. Such testing could be accomplished *concurrently* on the same physical computer using VMware. In this case, the programmer could test the application on a host operating system and on three guest operating systems with each system running as a separate virtual machine.

The architecture of such a system is shown in Figure 2.16. In this scenario, Linux is running as the host operating system; FreeBSD, Windows NT, and Windows XP are running as guest operating systems. The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

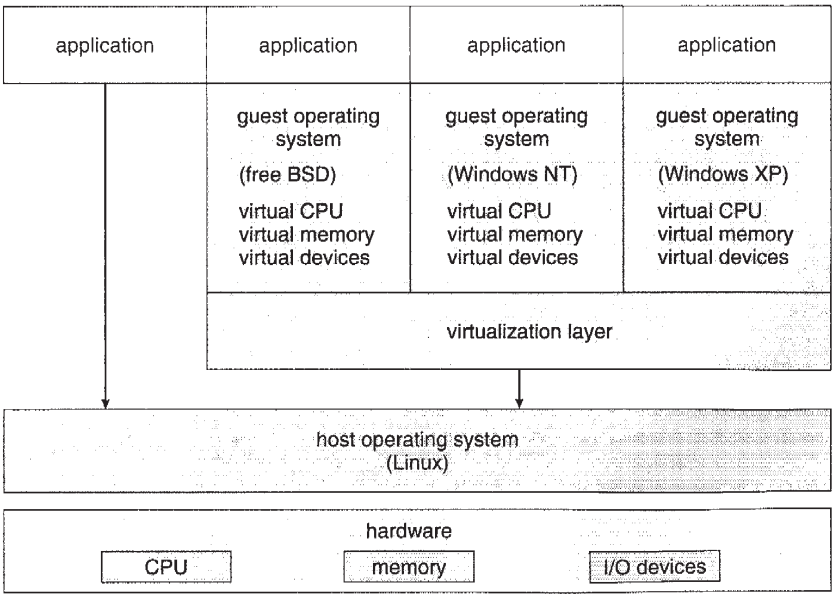



**Figure 2.16** VMware architecture.

### 2.8.3.2 The Java Virtual Machine

Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995. In addition to a language specification and a large API library, Java also provides a specification for a Java virtual machine—or JVM.

Java objects are specified with the `class` construct; a Java program consists of one or more classes. For each Java class, the compiler produces an architecture-neutral **bytecode** output (`.class`) file that will run on any implementation of the JVM.

The JVMis a specification for an abstract computer. It consists of a **class loader** and a Java interpreter that executes the architecture-neutral bytecodes, as diagrammed in Figure 2.17. The class loader loads the compiled `.class` files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the `.class` file is valid Java bytecode and does not overflow or underflow the stack. It also ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access. If the class passes verification, it is run by the Java interpreter. The JVM also automatically manages memory by performing **garbage collection**—the practice of reclaiming memory from objects no longer in use and returning it to the system. Much research focuses on garbage collection algorithms for increasing the performance of Java programs in the virtual machine.

The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or Mac OS X, or as part of a web browser. Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs. If the JVM is implemented in software, the Java interpreter interprets the bytecode operations one at a time. A faster software technique is to use a **just-in-time (JIT)** compiler. Here, the first time a Java method is invoked, the bytecodes for the method are turned into native machine language for the host system. These operations are then cached so that subsequent invocations of a method are performed using the native machine instructions and the bytecode operations need not be interpreted all over again. A technique that is potentially even faster is to run the JVM in hardware on a special Java chip that executes the Java bytecode operations as native code, thus bypassing the need for either a software interpreter or a just-in-time compiler.

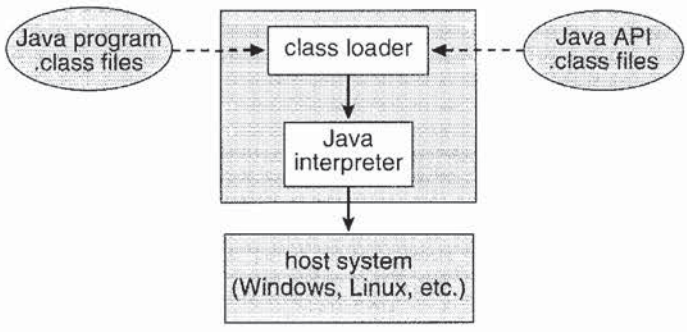



**Figure 2.17** The Java virtual machine.

## THE .NET FRAMEWORK

The .NET Framework is a collection of technologies, including a set of class libraries, and an execution environment that come together to provide a platform for developing software. This platform allows programs to be written to target the .NET Framework instead of a specific architecture. A program written for the .NET Framework need not worry about the specifics of the hardware or the operating system on which it will run. Thus, any architecture implementing .NET will be able to successfully execute the program. This is because the execution environment abstracts these details and provides a virtual machine as an intermediary between the executing program and the underlying architecture.

At the core of the .NET Framework is the Common Language Runtime (CLR). The CLR is the implementation of the .NET virtual machine. It provides an environment for execution of programs written in any of the languages targeted at the .NET Framework. Programs written in languages such as C# (pronounced *C-sharp*) and VB.NET are compiled into an intermediate, architecture-independent language called Microsoft Intermediate Language (MS-IL). These compiled files, called assemblies, include MS-IL instructions and metadata. They have a file extension of either .EXE or .DLL. Upon execution of a program, the CLR loads assemblies into what is known as the **Application Domain**. As instructions are requested by the executing program, the CLR converts the MS-IL instructions inside the assemblies into native code that is specific to the underlying architecture using just-in-time compilation. Once instructions have been converted to native code, they are kept and will continue to run as native code for the CPU. The architecture of the CLR for the .NET framework is shown in Figure 2.18.

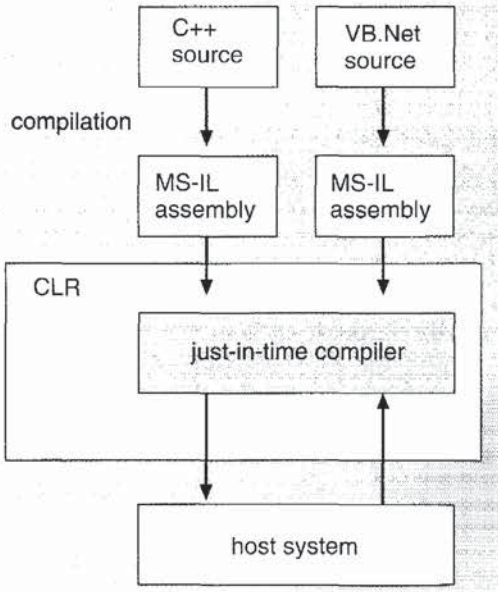



**Figure 2.18** Architecture of the CLR for the .NET Framework.

## 2.9 Operating-System Generation

It is possible to design, code, and implement an operating system specifically for one machine at one site. More commonly, however, operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations. The system must then be configured or generated for each specific computer site, a process sometimes known as **system generation (SYSGEN)**.

The operating system is normally distributed on disk or CD-ROM. To generate a system, we use a special program. The SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system, or probes the hardware directly to determine what components are there. The following kinds of information must be determined.

- What CPU is to be used? What options (extended instruction sets, floating-point arithmetic, and so on) are installed? For multiple CPU systems, each CPU must be described.
- How much memory is available? Some systems will determine this value themselves by referencing memory location after memory location until an "illegal address" fault is generated. This procedure defines the final legal address and hence the amount of available memory.
- What devices are available? The system will need to know how to address each device (the device number), the device interrupt number, the device's type and model, and any special device characteristics.
- What operating-system options are desired, or what parameter values are to be used? These options or values might include how many buffers of which sizes should be used, what type of CPU-scheduling algorithm is desired, what the maximum number of processes to be supported is, and so on.

Once this information is determined, it can be used in several ways. At one extreme, a system administrator can use it to modify a copy of the source code of the operating system. The operating system then is completely compiled. Data declarations, initializations, and constants, along with conditional compilation, produce an output object version of the operating system that is tailored to the system described.

At a slightly less tailored level, the system description can cause the creation of tables and the selection of modules from a precompiled library. These modules are linked together to form the generated operating system. Selection allows the library to contain the device drivers for all supported I/O devices, but only those needed are linked into the operating system. Because the system is not recompiled, system generation is faster, but the resulting system may be overly general.

At the other extreme, it is possible to construct a system that is completely table driven. All the code is always part of the system, and selection occurs at execution time, rather than at compile or link time. System generation involves simply creating the appropriate tables to describe the system.

The major differences among these approaches are the size and generality of the generated system and the ease of modification as the hardware configuration changes. Consider the cost of modifying the system to support a newly acquired graphics terminal or another disk drive. Balanced against that cost, of course, is the frequency (or infrequency) of such changes.

## 2.10 System Boot

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? The procedure of starting a computer by loading the kernel is known as *booting* the system. On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of **read-only memory** (ROM), because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot be infected by a computer virus.

The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system.

Some systems—such as cellular phones, PDAs, and game consoles—store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation. A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips. Some systems resolve this problem by using **erasable programmable read-only memory** (EPROM), which is read-only except when explicitly given a command to become writable. All forms of ROM are also known as **firmware**, since their characteristics fall somewhere between those of hardware and those of software. A problem with firmware in general is that executing code there is slower than executing code in RAM. Some systems store the operating system in firmware and copy it to RAM for fast execution. A final issue with firmware is that it is relatively expensive, so usually only small amounts are available.

For large operating systems (including most general-purpose operating systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader is stored in firmware, and the operating system is on disk. In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that **boot block**. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it fits in a single disk block) and only knows the address on disk and length of the

remainder of the bootstrap program. All of the disk-bound bootstrap, and the operating system itself, can be easily changed by writing new versions to disk. A disk that has a boot partition (more on that in section 12.5.1) is called a **boot disk** or **system disk**.

Now that the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution. It is only at this point that the system is said to be **running**.

## 2.11 Summary

Operating systems provide a number of services. At the lowest level, system calls allow a running program to make requests from the operating system directly. At a higher level, the command interpreter or shell provides a mechanism for a user to issue a request without writing a program. Commands may come from files during batch-mode execution or directly from a terminal when in an interactive or time-shared mode. System programs are provided to satisfy many common user requests.

The types of requests vary according to level. The system-call level must provide the basic functions, such as process control and file and device manipulation. Higher-level requests, satisfied by the command interpreter or system programs, are translated into a sequence of system calls. System services can be classified into several categories: program control, status requests, and I/O requests. Program errors can be considered implicit requests for service.

Once the system services are defined, the structure of the operating system can be developed. Various tables are needed to record the information that defines the state of the computer system and the status of the system's jobs.

The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. The type of system desired is the foundation for choices among various algorithms and strategies that will be needed.

Since an operating system is large, modularity is important. Designing a system as a sequence of layers or using a microkernel is considered a good technique. The virtual-machine concept takes the layered approach and treats both the kernel of the operating system and the hardware as though they were hardware. Even other operating systems may be loaded on top of this virtual machine.

Throughout the entire operating-system design cycle, we must be careful to separate policy decisions from implementation details (mechanisms). This separation allows maximum flexibility if policy decisions are to be changed later.

Operating systems are now almost always written in a systems-implementation language or in a higher-level language. This feature improves their implementation, maintenance, and portability. To create an operating system for a particular machine configuration, we must perform system generation.

For a computer system to begin running, the CPU must initialize and start executing the bootstrap program in firmware. The bootstrap can execute the operating system directly if the operating system is also in the firmware, or it can complete a sequence in which it loads progressively smarter programs

from firmware and disk until the operating system itself is loaded into memory and executed.

## Exercises

**2.1** The services and functions provided by an operating system can be divided into two main categories. Briefly describe the two categories and discuss how they differ.

**2.2** List five services provided by an operating system that are designed to make it more convenient for users to use the computer system. In what cases it would be impossible for user-level programs to provide these services? Explain.

**2.3** Describe three general methods for passing parameters to the operating system.

**2.4** Describe how you could obtain a statistical profile of the amount of time spent by a program executing different sections of its code. Discuss the importance of obtaining such a statistical profile.

**2.5** What are the five major activities of an operating system with regard to file management?

**2.6** What are the advantages and disadvantages of using the same system-call interface for manipulating both files and devices?

**2.7** What is the purpose of the command interpreter? Why is it usually separate from the kernel? Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?

**2.8** What are the two models of interprocess communication? What are the strengths and weaknesses of the two approaches?

**2.9** Why is the separation of mechanism and policy desirable?

**2.10** Why does Java provide the ability to call from a Java program native methods that are written in, say, C or C++? Provide an example of a situation in which a native method is useful.

**2.11** It is sometimes difficult to achieve a layered approach if two components of the operating system are dependent on each other. Identify a scenario in which it is unclear how to layer two system components that require tight coupling of their functionalities.

**2.12** What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?

**2.13** In what ways is the modular kernel approach similar to the layered approach? In what ways does it differ from the layered approach?

**2.14** What is the main advantage for an operating-system designer of using a virtual-machine architecture? What is the main advantage for a user?

**2.15** Why is a just-in-time compiler useful for executing Java programs?

**2.16** What is the relationship between a guest operating system and a host operating system in a system like VMware? What factors need to be considered in choosing the host operating system?

**2.17** The experimental Synthesis operating system has an assembler incorporated in the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended to make building the operating system easier. Discuss the pros and cons of the Synthesis approach to kernel design and system-performance optimization.

**2.18** In Section 2.3, we described a program that copies the contents of one file to a destination file. This program works by first prompting the user for the name of the source and destination files. Write this program using either the Windows32 or POSIX API. Be sure to include all necessary error checking, including ensuring that the source file exists. Once you have correctly designed and tested the program, if you used a system that supports it, run the program using a utility that traces system calls. Linux systems provide the `ptrace` utility, and Solaris systems use the `truss` or `dtrace` command. On Mac OS X, the `ktrace` facility provides similar functionality.

## Project—Adding a System Call to the Linux Kernel

In this project, you will study the system call interface provided by the Linux operating system and how user programs communicate with the operating system kernel via this interface. Your task is to incorporate a new system call into the kernel, thereby expanding the functionality of the operating system.

### Getting Started

A user-mode procedure call is performed by passing arguments to the called procedure either on the stack or through registers, saving the current state and the value of the program counter, and jumping to the beginning of the code corresponding to the called procedure. The process continues to have the same privileges as before.

System calls appear as procedure calls to user programs, but result in a change in execution context and privileges. In Linux on the Intel 386 architecture, a system call is accomplished by storing the system call number into the EAX register, storing arguments to the system call in other hardware registers, and executing a trap instruction (which is the INT 0x80 assembly instruction). After the trap is executed, the system call number is used to index into a table of code pointers to obtain the starting address for the handler code implementing the system call. The process then jumps to this address and the privileges of the process are switched from user to kernel mode. With the expanded privileges, the process can now execute kernel code that might

include privileged instructions that cannot be executed in user mode. The kernel code can then perform the requested services such as interacting with I/O devices, perform process management and other such activities that cannot be performed in user mode.

The system call numbers for recent versions of the Linux kernel are listed in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. (For instance, `__NR_close`, which corresponds to the system call `close()` that is invoked for closing a file descriptor, is defined as value 6.) The list of pointers to system call handlers is typically stored in the file `/usr/src/linux-2.x/arch/i386/kernel/entry.S` under the heading `ENTRY(sys_call_table)`. Notice that `sys_close` is stored at entry numbered 6 in the table to be consistent with the system call number defined in `unistd.h` file. (The keyword `.long` denotes that the entry will occupy the same number of bytes as a data value of type `long`.)

### Building a New Kernel

Before adding a system call to the kernel, you must familiarize yourself with the task of building the binary for a kernel from its source code and booting the machine with the newly built kernel. This activity comprises the following tasks, some of which are dependent on the particular installation of the Linux operating system.

- Obtain the kernel source code for the Linux distribution. If the source code package has been previously installed on your machine, the corresponding files might be available under `/usr/src/linux` or `/usr/src/linux-2.x` (where the suffix corresponds to the kernel version number). If the package has not been installed earlier, it can be downloaded from the provider of your Linux distribution or from `http://www.kernel.org`.
- Learn how to configure, compile, and install the kernel binary. This will vary between the different kernel distributions, but some typical commands for building the kernel (after entering the directory where the kernel source code is stored) include:
  - `make xconfig`
  - `make dep`
  - `make bzImage`
- Add a new entry to the set of bootable kernels supported by the system. The Linux operating system typically uses utilities such as `lilo` and `grub` to maintain a list of bootable kernels, from which the user can choose during machine boot-up. If your system supports `lilo`, add an entry to `lilo.conf`, such as:

```
image=/boot/bzImage.mykernel
label=mykernel
root=/dev/hda5
read-only
```

  where `/boot/bzImage.mykernel` is the kernel image and `mykernel` is

the label associated with the new kernel allowing you to choose it during bootup process. By performing this step, you have the option of either booting a new kernel or booting the unmodified kernel if the newly built kernel does not function properly.

### Extending Kernel Source

You can now experiment with adding a new file to the set of source files used for compiling the kernel. Typically, the source code is stored in the `/usr/src/linux-2.x/kernel` directory, although that location may differ in your Linux distribution. There are two options for adding the system call. The first is to add the system call to an existing source file in this directory. A second option is to create a new file in the source directory and modify `/usr/src/linux-2.x/kernel/Makefile` to include the newly created file in the compilation process. The advantage of the first approach is that by modifying an existing file that is already part of the compilation process, the Makefile does not require modification.

### Adding a System Call to the Kernel

Now that you are familiar with the various background tasks corresponding to building and booting Linux kernels, you can begin the process of adding a new system call to the Linux kernel. In this project, the system call will have limited functionality; it will simply transition from user mode to kernel mode, print a message that is logged with the kernel messages, and transition back to user mode. We will call this the *helloworld* system call. While it has only limited functionality, it illustrates the system call mechanism and sheds light on the interaction between user programs and the kernel.

- Create a new file called `helloworld.c` to define your system call. Include the header files `linux/linkage.h` and `linux/kernel.h`. Add the following code to this file:

```
#include <linux/linkage.h>
#include <linux/kernel.h>
asmlinkage int sys_helloworld() {
   printk(KERN_EMERG "hello world!");

   return 1;
}
```

This creates a system call with the name `sys_helloworld()`. If you choose to add this system call to an existing file in the source directory, all that is necessary is to add the `sys_helloworld()` function to the file you choose. `asmlinkage` is a remnant from the days when Linux used both C++ and C code and is used to indicate that the code is written in C. The `printk()` function is used to print messages to a kernel log file and therefore may only be called from the kernel. The kernel messages specified in the parameter to `printk()` are logged in the file `/var/log/kernel/warnings`. The function prototype for the `printk()` call is defined in `/usr/include/linux/kernel.h`.

- Define a new system call number for `__NR_helloworld` in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. A user program can use this number to identify the newly added system call. Also be sure to increment the value for `__NR_syscalls`, which is also stored in the same file. This constant tracks the number of system calls currently defined in the kernel.
- Add an entry `.long sys_helloworld` to the `sys_call_table` defined in `/usr/src/linux-2.x/arch/i386/kernel/entry.S` file. As discussed earlier, the system call number is used to index into this table to find the position of the handler code for the invoked system call.
- Add your file `helloworld.c` to the Makefile (if you created a new file for your system call.) Save a copy of your old kernel binary image (in case there are problems with your newly created kernel.) You can now build the new kernel, rename it to distinguish it from the unmodified kernel, and add an entry to the loader configuration files (such as `lilo.conf`). After completing these steps, you may now boot either the old kernel or the new kernel that contains your system call inside it.

### Using the System Call From a User Program

When you boot with the new kernel it will support the newly defined system call; it is now simply a matter of invoking this system call from a user program. Ordinarily, the standard C library supports an interface for system calls defined for the Linux operating system. As your new system call is not linked into the standard C library, invoking your system call will require manual intervention.

As noted earlier, a system call is invoked by storing the appropriate value into a hardware register and performing a trap instruction. Unfortunately, these are low-level operations that cannot be performed using C language statements and instead require assembly instructions. Fortunately, Linux provides macros for instantiating wrapper functions that contain the appropriate assembly instructions. For instance, the following C program uses the `_syscall0()` macro to invoke the newly defined system call:

```
#include <linux/errno.h>
#include <sys/syscall.h>
#include <linux/unistd.h>

_syscall0(int, helloworld);

main()
{
   helloworld();
}
```

- The `_syscall0` macro takes two arguments. The first specifies the type of the value returned by the system call; the second argument is the name of the system call. The name is used to identify the system call number that is stored in the hardware register before the trap instruction is executed.

If your system call requires arguments, then a different macro (such as `_syscall0`, where the suffix indicates the number of arguments) could be used to instantiate the assembly code required for performing the system call.

- Compile and execute the program with the newly built kernel. There should be a message "`hello world!`" in the kernel log file `/var/log/kernel/warnings` to indicate that the system call has executed.

As a next step, consider expanding the functionality of your system call. How would you pass an integer value or a character string to the system call and have it be printed into the kernel log file? What are the implications for passing pointers to data stored in the user program's address space as opposed to simply passing an integer value from the user program to the kernel using hardware registers?

## Bibliographical Notes

Dijkstra [1968] advocated the layered approach to operating-system design. Brinch-Hansen [1970] was an early proponent of constructing an operating system as a kernel (or nucleus) on which more complete systems can be built.

System instrumentation and dynamic tracing are described in Tamches and Miller [1999]. DTrace is discussed in Cantrill et al. [2004]. Cheung and Loong [1995] explored issues of operating-system structure from microkernel to extensible systems.

MS-DOS, Version 3.1, is described in Microsoft [1986]. Windows NT and Windows 2000 are described by Solomon [1998] and Solomon and Russinovich [2000]. BSD UNIX is described in McKusick et al. [1996]. Bovet and Cesati [2002] cover the Linux kernel in detail. Several UNIX systems—including Mach—are treated in detail in Vahalia [1996]. Mac OS X is presented at http://www.apple.com/macosx. The experimental Synthesis operating system is discussed by Massalin and Pu [1989]. Solaris is fully described in Mauro and McDougall [2001].

The first operating system to provide a virtual machine was the CP/67 on an IBM 360/67. The commercially available IBM VM/370 operating system was derived from CP/67. Details regarding Mach, a microkernel-based operating system, can be found in Young et al. [1987]. Kaashoek et al. [1997] present details regarding exokernel operating systems, where the architecture separates management issues from protection, thereby giving untrusted software the ability to exercise control over hardware and software resources.

The specifications for the Java language and the Java virtual machine are presented by Gosling et al. [1996] and by Lindholm and Yellin [1999], respectively. The internal workings of the Java virtual machine are fully described by Venners [1998]. Golm et al. [2002] highlight the JX operating system; Back et al. [2000] cover several issues in the design of Java operating systems. More information on Java is available on the Web at http://www.javasoft.com. Details about the implementation of VMware can be found in Sugerman et al. [2001].

Part Two

# Process Management

A *process* can be thought of as a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices—to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

A process is the unit of work in most systems. Systems consist of a collection of processes: Operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

Although traditionally a process contained only a single *thread* of control as it ran, most modern operating systems now support processes that have multiple threads.

The operating system is responsible for the following activities in connection with process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.




# Processes

Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a **process**, which is a program in execution. A process is the unit of work in a modern time-sharing system.

The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself. A system therefore consists of a collection of processes: operating-system processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive.

## CHAPTER OBJECTIVES

- To introduce the notion of a process — a program in execution, which forms the basis of all computation.
- To describe the various features of processes, including scheduling, creation and termination, and communication.
- To describe communication in client–server systems.

## 3.1 Process Concept

A question that arises in discussing operating systems involves what to call all the CPU activities. A batch system executes *jobs*, whereas a time-shared system has *user programs*, or *tasks*. Even on a single-user system such as Microsoft Windows, a user may be able to run several programs at one time: a word processor, a web browser, and an e-mail package. Even if the user can execute

only one program at a time, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them *processes*.

The terms *job* and *process* are used almost interchangeably in this text. Although we personally prefer the term *process*, much of operating-system theory and terminology was developed during a time when the major activity of operating systems was job processing. It would be misleading to avoid the use of commonly accepted terms that include the word *job* (such as *job scheduling*) simply because *process* has superseded *job*.

### 3.1.1 The Process

Informally, as mentioned earlier, a process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure 3.1.

We emphasize that a program by itself is not a process; a program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an **executable file**), whereas a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in `prog.exe` or `a.out`.)

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance,

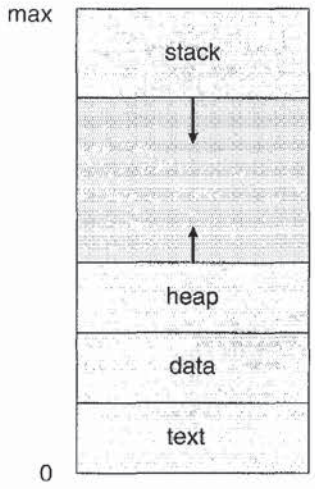



**Figure 3.1** Process in memory.

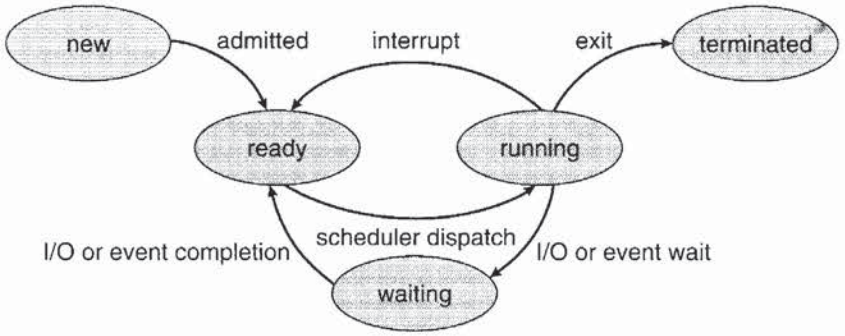



**Figure 3.2** Diagram of process state.

several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary. It is also common to have a process that spawns many processes as it runs. We discuss such matters in Section 3.4.

### 3.1.2 Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready**. The process is waiting to be assigned to a processor.
- **Terminated**. The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting*, however. The state diagram corresponding to these states is presented in Figure 3.2.

### 3.1.3 Process Control Block

Each process is represented in the operating system by a **process control block** (PCB)—also called a *task control block*. A PCB is shown in Figure 3.3. It contains many pieces of information associated with a specific process, including these:

- **Process state**. The state may be new, ready, running, waiting, halted, and so on.

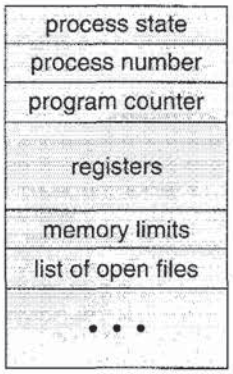



**Figure 3.3** Process control block (PCB).

- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 3.4).
- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 describes process scheduling.)
- **Memory-management information**. This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 8).
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for any information that may vary from process to process.

### 3.1.4 Threads

The process model discussed so far has implied that a process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at one time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Many modern operating systems have extended the process concept to allow a process to have multiple

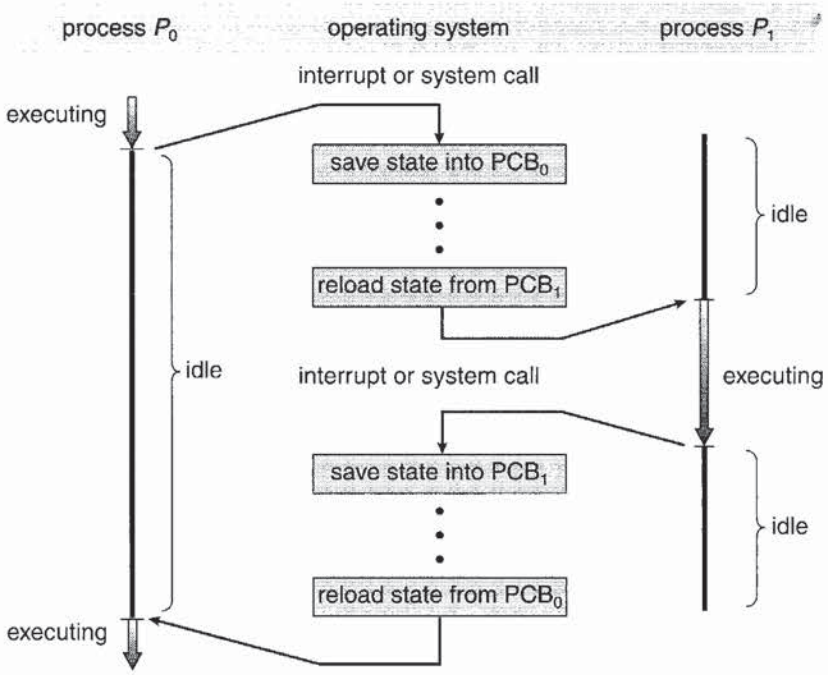



**Figure 3.4** Diagram showing CPU switch from process to process.

threads of execution and thus to perform more than one task at a time. Chapter 4 explores multithreaded processes in detail.

## 3.2 Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

### 3.2.1 Scheduling Queues

As processes enter the system, they are put into a **job queue**, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.

## PROCESS REPRESENTATION IN LINUX

The process control block in the Linux operating system is represented by the C structure `task_struct`. This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory management information, list of open files, and pointers to the process's parent and any of its children. (A process's *parent* is the process that created it; its *children* are any processes that it creates.) Some of these fields include:

```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

For example, the state of a process is represented by the field `long state` in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`, and the kernel maintains a pointer — `current` — to the process currently executing on the system. This is shown in Figure 3.5.

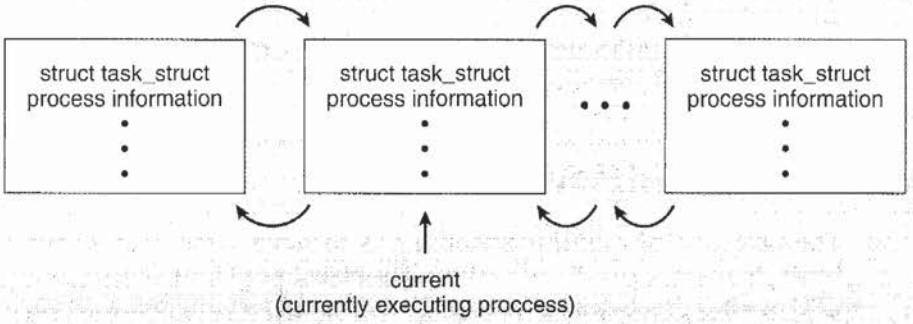



**Figure 3.5** Active processes in Linux.

As an illustration of how the kernel might manipulate one of the fields in the `task_struct` for a specified process, let's assume the system would like to change the state of the process currently running to the value `new_state`. If `current` is a pointer to the process currently executing, its state is changed with the following:

```
current->state = new_state;
```

Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue (Figure 3.6).

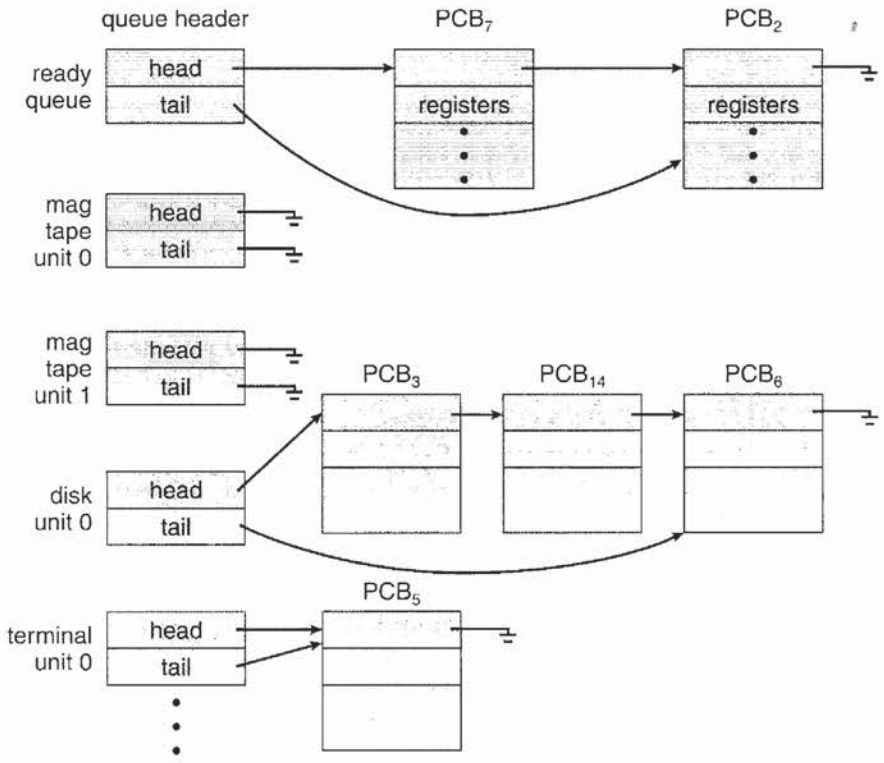



**Figure 3.6** The ready queue and various I/O device queues.

A common representation for a discussion of process scheduling is a **queueing diagram**, such as that in Figure 3.7. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new subprocess and wait for the subprocess's termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

### 3.2.2 Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes

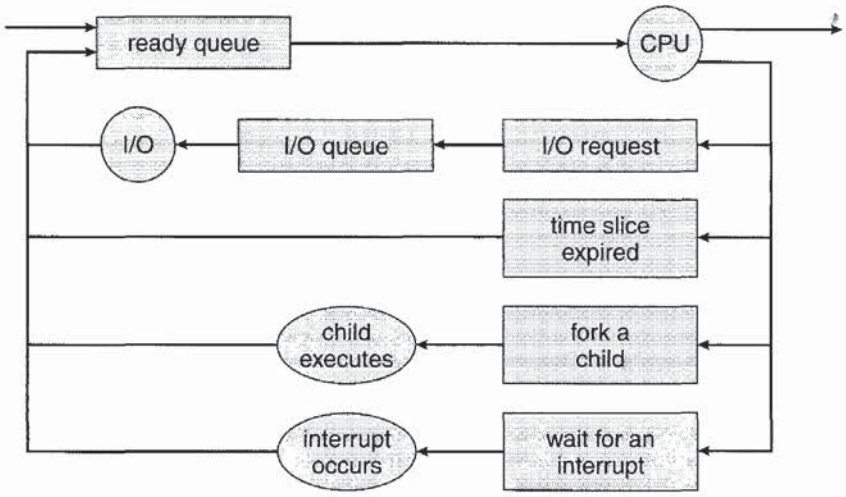



**Figure 3.7** Queueing-diagram representation of process scheduling.

from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution. The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then $10/(100 + 10) = 9$ percent of the CPU is being used (wasted) simply for scheduling the work.

The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either I/O bound or CPU bound. An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations. A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations. It is important that the long-term scheduler select a good **process mix** of I/O-bound and CPU-bound

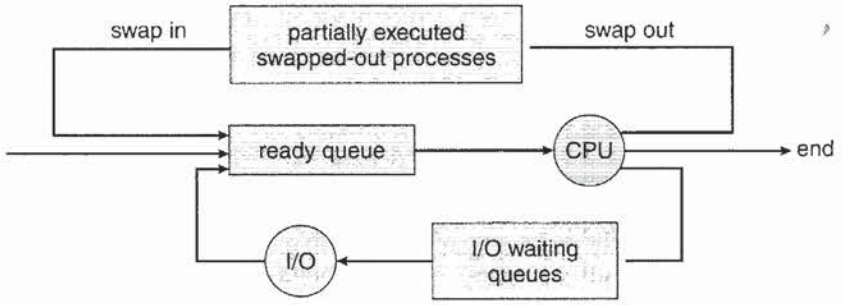



**Figure 3.8** Addition of medium-term scheduling to the queueing diagram.

processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.

On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler. The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users. If the performance declines to unacceptable levels on a multiuser system, some users will simply quit.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is diagrammed in Figure 3.8. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. Swapping is discussed in Chapter 8.

### 3.2.3 Context Switch

As mentioned in 1.2.1, interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current **context** of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process; it includes the value of the CPU registers, the process state (see Figure 3.2), and memory-management information. Generically, we perform a **state save** of the current state of the CPU, be it in kernel or user mode, and then a **state restore** to resume operations.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds are a few milliseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the more work must be done during a context switch. As we will see in Chapter 8, advanced memory-management techniques may require extra data to be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system.

## 3.3 Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

### 3.3.1 Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems (including UNIX and the Windows family of operating systems) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number. Figure 3.9 illustrates a typical process tree for the Solaris operating system, showing the name of each process and its pid. In Solaris, the process at the top of the tree is the `sched` process, with pid of 0. The `sched` process creates several children processes—including `pageout` and `fsflush`. These processes are responsible for managing memory and file systems. The `sched` process also creates the `init` process, which serves as the root parent process for all user processes. In Figure 3.9, we see two children of `init`—`inetd` and `dtlogin`. `inetd` is responsible for networking services such as `telnet` and `ftp`; `dtlogin` is the process representing a user login screen. When a user logs in, `dtlogin` creates an X-windows session (`Xsession`), which in turns creates the `sdt_shel` process. Below `sdt_shel`, a

user's command-line shell—the C-shell or `csh`—is created. It is this command-line interface where the user then invokes various child processes, such as the `ls` and `cat` commands. We also see a `csh` process with pid of 7778 representing a user who has logged onto the system using `telnet`. This user has started the Netscape browser (pid of 7785) and the emacs editor (pid of 8105).

On UNIX, a listing of processes can be obtained using the `ps` command. For example, entering the command `ps -el` will list complete information for all processes currently active in the system. It is easy to construct a process tree similar to what is shown in Figure 3.9 by recursively tracing parent processes all the way to the `init` process.

In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.

In addition to the various physical and logical resources that a process obtains when it is created, initialization data (input) may be passed along by the parent process to the child process. For example, consider a process whose function is to display the contents of a file—say, *img.jpg*—on the screen of a

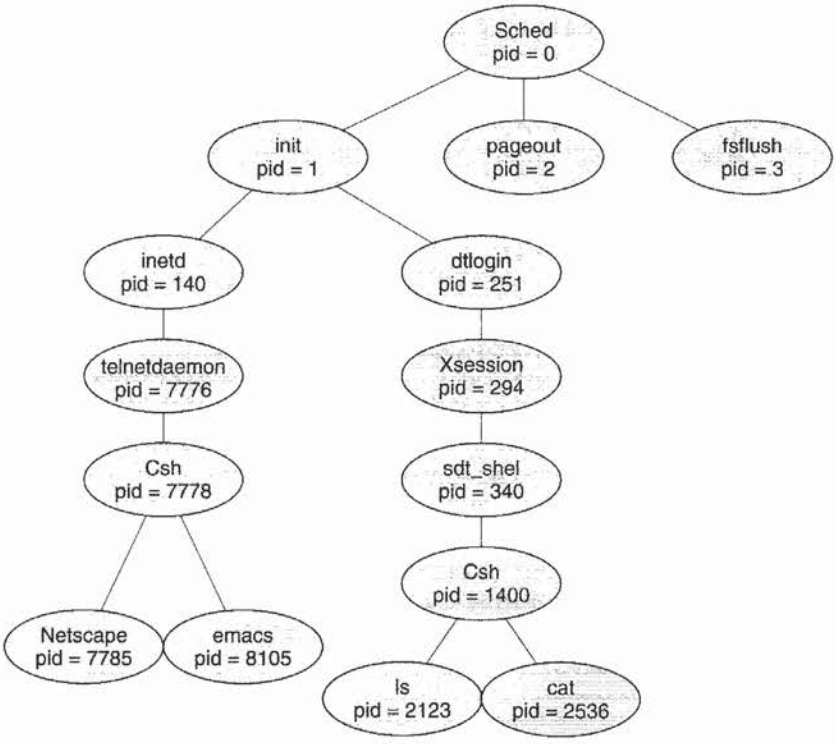



**Figure 3.9** A tree of processes on a typical Solaris system.

terminal. When it is created, it will get, as an input from its parent process, the name of the file *img.jpg*, and it will use that file name, open the file, and write the contents out. It may also get the name of the output device. Some operating systems pass resources to child processes. On such a system, the new process may get two open files, *img.jpg* and the terminal device, and may simply transfer the datum between the two.

When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, as we've seen, each process is identified by its process identifier,

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {/* error occurred */
       fprintf(stderr, "Fork Failed");
       exit(-1);
    }
    else if (pid == 0) {/* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else {/* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
       exit(0);
    }
}
```

**Figure 3.10** C program forking a separate process.

which is a unique integer. A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the `fork()`, with one difference: The return code for the `fork()` is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the `exec()` system call is used after a `fork()` system call by one of the two processes to replace the process's memory space with a new program. The `exec()` system call loads a binary file into memory (destroying the memory image of the program containing the `exec()` system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a `wait()` system call to move itself off the ready queue until the termination of the child.

The C program shown in Figure 3.10 illustrates the UNIX system calls previously described. We now have two different processes running a copy of the same program. The value of pid for the child process is zero; that for the parent is an integer value greater than zero. The child process overlays its address space with the UNIX command `/bin/ls` (used to get a directory listing) using the `execlp()` system call (`execlp()` is a version of the `exec()` system call). The parent waits for the child process to complete with the `wait()` system call. When the child process completes (by either implicitly or explicitly invoking `exit()`) the parent process resumes from the call to `wait()`, where it completes using the `exit()` system call. This is also illustrated in Figure 3.11.

As an alternative example, we next consider process creation in Windows. Processes are created in the Win32 API using the `CreateProcess()` function, which is similar to `fork()` in that a parent creates a new child process. However, whereas `fork()` has the child process inheriting the address space of its parent, `CreateProcess()` requires loading a specified program into the address space of the child process at process creation. Furthermore, whereas `fork()` is passed no parameters, `CreateProcess()` expects no fewer than ten parameters.

The C program shown in Figure 3.12 illustrates the `CreateProcess()` function, which creates a child process that loads the application `mspaint.exe`. We opt for many of the default values of the ten parameters passed to `CreateProcess()`. Readers interested in pursuing the details on process creation and management in the Win32 API are encouraged to consult the bibliographical notes at the end of this chapter.

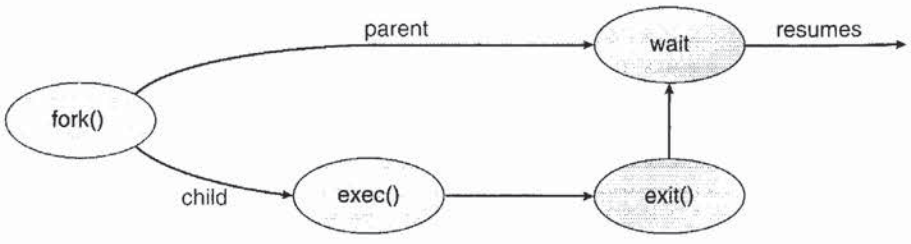



**Figure 3.11** Process creation.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // use command line
     "C:\\WINDOWS\\system32\\mspaint.exe", // command line
     NULL, // don't inherit process handle
     NULL, // don't inherit thread handle
     FALSE, // disable handle inheritance
     0, // no creation flags
     NULL, // use parent's environment block
     NULL, // use parent's existing directory
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

**Figure 3.12** Creating a separate process using the Win32 API.

Two parameters passed to `CreateProcess()` are instances of the STARTUPINFO and PROCESS_INFORMATION structures. STARTUPINFO specifies many properties of the new process, such as window size and appearance and handles to standard input and output files. The PROCESS_INFORMATION structure contains a handle and the identifiers to the newly created process and its thread. We invoke the `ZeroMemory()` function to allocate memory for each of these structures before proceeding with `CreateProcess()`.

The first two parameters passed to `CreateProcess()` are the application name and command line parameters. If the application name is NULL (which in this case it is), the command line parameter specifies the application to load. In this instance we are loading the Microsoft Windows *mspaint.exe*

application. Beyond these two initial parameters, we use the default parameters for inheriting process and thread handles as well as specifying no creation flags. We also use the parent's existing environment block and starting directory. Last, we provide two pointers to the STARTUPINFO and PROCESS_INFORMATION structures created at the beginning of the program. In Figure 3.10, the parent process waits for the child to complete by invoking the `wait()` system call. The equivalent of this in Win32 is `WaitForSingleObject()`, which is passed a handle of the child process—`pi.hProcess`— that it is waiting for to complete. Once the child process exits, control returns from the `WaitForSingleObject()` function in the parent process.

### 3.3.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, `TerminateProcess()` in Win32). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs. Note that a parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems, including VMS, do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

To illustrate process execution and termination, consider that, in UNIX, we can terminate a process by using the `exit()` system call; its parent process may wait for the termination of a child process by using the `wait()` system call. The `wait()` system call returns the process identifier of a terminated child so that the parent can tell which of its possibly many children has terminated. If the parent terminates, however, all its children have assigned as their new parent the `init` process. Thus, the children still have a parent to collect their status and execution statistics.

## 3.4 Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is **independent** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

- **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads, as we discussed in Chapter 2.
- **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an **interprocess communication** (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: (1) **shared memory** and (2) **message passing**. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure 3.13.

Both of the models just discussed are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer. Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In contrast, in shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required. In the remainder of this section, we explore each of these IPC models in more detail.

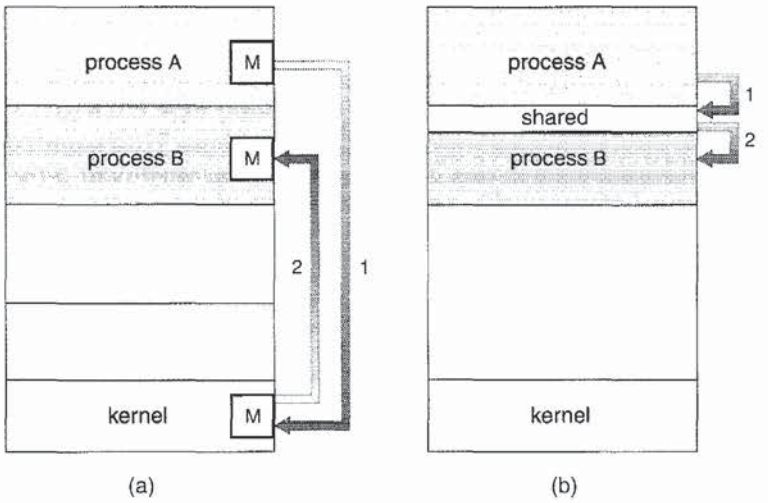



**Figure 3.13** Communications models. (a) Message passing. (b) Shared memory.

### 3.4.1 Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader. The producer–consumer problem also provides a useful metaphor for the client–server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must

be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Let's look more closely at how the bounded buffer can be used to enable processes to share memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {
   . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers: `in` and `out`. The variable `in` points to the next free position in the buffer; `out` points to the first full position in the buffer. The buffer is empty when `in == out`; the buffer is full when `((in + 1) % BUFFER_SIZE) == out`.

The code for the producer and consumer processes is shown in Figures 3.14 and 3.15, respectively. The producer process has a local variable `nextProduced` in which the new item to be produced is stored. The consumer process has a local variable `nextConsumed` in which the item to be consumed is stored.

This scheme allows at most BUFFER_SIZE – 1 items in the buffer at the same time. We leave it as an exercise for you to provide a solution where BUFFER_SIZE items can be in the buffer at the same time. In Section 3.5.1, we illustrate the POSIX API for shared memory.

One issue this illustration does not address concerns the situation in which both the producer process and the consumer process attempt to access the shared buffer concurrently. In Chapter 6, we discuss how synchronization among cooperating processes can be implemented effectively in a shared-memory environment.

```
item nextProduced;

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

**Figure 3.14** The producer process.

```
item nextConsumed;

while (true) {
     while (in == out)
        ; // do nothing

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
}
```

**Figure 3.15** The consumer process.

### 3.4.2 Message-Passing Systems

In Section 3.4.1, we showed how cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, a **chat** program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations: `send`(message) and `receive`(message). Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating system design.

If processes $P$ and $Q$ want to communicate, they must send messages to and receive messages from each other; a **communication link** must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, which are covered in Chapter 16) but rather with its logical implementation. Here are several methods for logically implementing a link and the `send()/receive()` operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

We look at issues related to each of these features next.

#### 3.4.2.1 Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under **direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the `send()` and `receive()` primitives are defined as:

- `send(P, message)`—Send a `message` to process `P`.
- `receive(Q, message)`—Receive a `message` from process `Q`.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

This scheme exhibits *symmetry* in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs *asymmetry* in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the `send()` and `receive()` primitives are defined as follows:

- `send(P, message)`—Send a `message` to process `P`.
- `receive(id, message)`—Receive a `message` from any process; the variable *id* is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such **hard-coding** techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next.

With **indirect communication**, the messages are sent to and received from **mailboxes**, or **ports**. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if the processes have a shared mailbox, however. The `send()` and `receive()` primitives are defined as follows:

- `send(A, message)`—Send a `message` to mailbox `A`.
- `receive(A, message)`—Receive a `message` from mailbox `A`.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

Now suppose that processes $P_1$, $P_2$, and $P_3$ all share mailbox $A$. Process $P_1$ sends a message to $A$, while both $P_2$ and $P_3$ execute a `receive()` from $A$. Which process will receive the message sent by $P_1$? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a `receive()` operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either $P_2$ or $P_3$, but not both, will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, *round robin* where processes take turns receiving messages). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

#### 3.4.2.2 Synchronization

Communication between processes takes place through calls to `send()` and `receive()` primitives. There are different design options for implementing

each primitive. Message passing may be either **blocking** or **nonblocking**—also known as **synchronous** and **asynchronous**.

- **Blocking send**. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send**. The sending process sends the message and resumes operation.
- **Blocking receive**. The receiver blocks until a message is available.
- **Nonblocking receive**. The receiver retrieves either a valid message or a null.

Different combinations of `send()` and `receive()` are possible. When both `send()` and `receive()` are blocking, we have a **rendezvous** between the sender and the receiver. The solution to the producer–consumer problem becomes trivial when we use blocking `send()` and `receive()` statements. The producer merely invokes the blocking `send()` call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes `receive()`, it blocks until a message is available.

Note that the concepts of synchronous and asynchronous occur frequently in operating-system I/O algorithms, as you will see throughout this text.

#### 3.4.2.3 Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- **Zero capacity**. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity**. The queue has finite length $n$; thus, at most $n$ messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The links capacity is finite , however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity**. The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

## 3.5 Examples of IPC Systems

In this section, we explore three different IPC systems. We first cover the POSIX API for shared memory and then discuss message passing in the Mach operating system. We conclude with Windows XP, which interestingly uses shared memory as a mechanism for providing certain types of message passing.

### 3.5.1 An Example: POSIX Shared Memory

Several IPC mechanisms are available for POSIX systems, including shared memory and message passing. Here, we explore the POSIX API for shared memory.

A process must first create a shared memory segment using the `shmget()` system call (`shmget()` is derived from SHared Memory GET). The following example illustrates the use of `shmget()`:

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

This first parameter specifies the key (or identifier) of the shared-memory segment. If this is set to IPC_PRIVATE, a new shared-memory segment is created. The second parameter specifies the size (in bytes) of the shared memory segment. Finally, the third parameter identifies the mode, which indicates how the shared-memory segment is to be used—that is, for reading, writing, or both. By setting the mode to S_IRUSR | S_IWUSR, we are indicating that the owner may read or write to the shared memory segment. A successful call to `shmget()` returns an integer identifier for the shared-memory segment. Other processes that want to use this region of shared memory must specify this identifier.

Processes that wish to access a shared-memory segment must attach it to their address space using the `shmat()` (SHared Memory ATtach) system call. The call to `shmat()` expects three parameters as well. The first is the integer identifier of the shared-memory segment being attached, and the second is a pointer location in memory indicating where the shared memory will be attached. If we pass a value of NULL, the operating system selects the location on the user's behalf. The third parameter identifies a flag that allows the shared-memory region to be attached in read-only or read-write mode; by passing a parameter of 0, we allow both reads and writes to the shared region.

The third parameter identifies a mode flag. If set, the mode flag allows the shared-memory region to be attached in read-only mode; if set to 0, the flag allows both reads and writes to the shared region. We attach a region of shared memory using `shmat()` as follows:

```
shared_memory = (char *) shmat(id, NULL, 0);
```

If successful, `shmat()` returns a pointer to the beginning location in memory where the shared-memory region has been attached.

Once the region of shared memory is attached to a process's address space, the process can access the shared memory as a routine memory access using the pointer returned from `shmat()`. In this example, `shmat()` returns a pointer to a character string. Thus, we could write to the shared-memory region as follows:

```
sprintf(shared_memory, "Writing to shared memory");
```

Other processes sharing this segment would see the updates to the shared-memory segment.

Typically, a process using an existing shared-memory segment first attaches the shared-memory region to its address space and then accesses (and possibly updates) the region of shared memory. When a process no longer requires access to the shared-memory segment, it detaches the segment from its address

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the identifier for the shared memory segment */
int segment_id;=
/* a pointer to the shared memory segment */
char* shared_memory;
/* the size (in bytes) of the shared memory segment */
const int size = 4096;

    /* allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* write a message to the shared memory segment */
    sprintf(shared_memory, "Hi there!");

    /* now print out the string from shared memory */
    printf("*%s\n", shared_memory);

    /* now detach the shared memory segment */
    shmdt(shared_memory);

    /* now remove the shared memory segment */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}
```

**Figure 3.16** C program illustrating POSIX shared-memory API.

space. To detach a region of shared memory, the process can pass the pointer of the shared-memory region to the `shmdt()` system call, as follows:

```
shmdt(shared_memory);
```

Finally, a shared-memory segment can be removed from the system with the `shmctl()` system call, which is passed the identifier of the shared segment along with the flag IPC_RMID.

The program shown in Figure 3.16 illustrates the POSIX shared-memory API discussed above. This program creates a 4,096-byte shared-memory segment. Once the region of shared memory is attached, the process writes the message `Hi There!` to shared memory. After outputting the contents of the updated memory, it detaches and removes the shared-memory region. We provide further exercises using the POSIX shared memory API in the programming exercises at the end of this chapter.

### 3.5.2 An Example: Mach

As an example of a message-based operating system, we next consider the Mach operating system, developed at Carnegie Mellon University. We introduced Mach in Chapter 2 as part of the Mac OS X operating system. The Mach kernel supports the creation and destruction of multiple tasks, which are similar to processes but have multiple threads of control. Most communication in Mach—including most of the system calls and all intertask information—is carried out by *messages*. Messages are sent to and received from mailboxes, called *ports* in Mach.

Even system calls are made by messages. When a task is created, two special mailboxes—the Kernel mailbox and the Notify mailbox—are also created. The Kernel mailbox is used by the kernel to communicate with the task. The kernel sends notification of event occurrences to the Notify port. Only three system calls are needed for message transfer. The `msg_send()` call sends a message to a mailbox. A message is received via `msg_receive()`. Remote procedure calls (RPCs) are executed via `msg_rpc()`, which sends a message and waits for exactly one return message from the sender. In this way, the RPC models a typical subroutine procedure call but can work between systems—hence the term *remote*.

The `port_allocate()` system call creates a new mailbox and allocates space for its queue of messages. The maximum size of the message queue defaults to eight messages. The task that creates the mailbox is that mailbox's owner. The owner is also allowed to receive from the mailbox. Only one task at a time can either own or receive from a mailbox, but these rights can be sent to other tasks if desired.

The mailbox has an initially empty queue of messages. As messages are sent to the mailbox, the messages are copied into the mailbox. All messages have the same priority. Mach guarantees that multiple messages from the same sender are queued in first-in, first-out (FIFO) order but does not guarantee an absolute ordering. For instance, messages from two senders may be queued in any order.

The messages themselves consist of a fixed-length header followed by a variable-length data portion. The header indicates the length of the message and includes two mailbox names. One mailbox name is the mailbox to which the message is being sent. Commonly, the sending thread expects a reply; so the mailbox name of the sender is passed on to the receiving task, which can use it as a "return address."

The variable part of a message is a list of typed data items. Each entry in the list has a type, size, and value. The type of the objects specified in the message is important, since objects defined by the operating system—such as ownership or receive access rights, task states, and memory segments—may be sent in messages.

The send and receive operations themselves are flexible. For instance, when a message is sent to a mailbox, the mailbox may be full. If the mailbox is not full, the message is copied to the mailbox, and the sending thread continues. If the mailbox is full, the sending thread has four options:

1. Wait indefinitely until there is room in the mailbox.
2. Wait at most $n$ milliseconds.

3. Do not wait at all but rather return immediately.
4. Temporarily cache a message. One message can be given to the operating system to keep, even though the mailbox to which it is being sent is full. When the message can be put in the mailbox, a message is sent back to the sender; only one such message to a full mailbox can be pending at any time for a given sending thread.

The final option is meant for server tasks, such as a line-printer driver. After finishing a request, such tasks may need to send a one-time reply to the task that had requested service; but they must also continue with other service requests, even if the reply mailbox for a client is full.

The receive operation must specify the mailbox or mailbox set from which a message is to be received. A **mailbox set** is a collection of mailboxes, as declared by the task, which can be grouped together and treated as one mailbox for the purposes of the task. Threads in a task can receive only from a mailbox or mailbox set for which the task has receive access. A `port_status()` system call returns the number of messages in a given mailbox. The receive operation attempts to receive from (1) any mailbox in a mailbox set or (2) a specific (named) mailbox. If no message is waiting to be received, the receiving thread can either wait at most *n* milliseconds or not wait at all.

The Mach system was especially designed for distributed systems, which we discuss in Chapters 16 through 18, but Mach is also suitable for single-processor systems, as evidenced by its inclusion in the Mac OS X system. The major problem with message systems has generally been poor performance caused by double copying of messages; the message is copied first from the sender to the mailbox and then from the mailbox to the receiver. The Mach message system attempts to avoid double-copy operations by using virtual-memory-management techniques (Chapter 9). Essentially, Mach maps the address space containing the sender's message into the receiver's address space. The message itself is never actually copied. This message-management technique provides a large performance boost but works for only intrasystem messages. The Mach operating system is discussed in an extra chapter posted on our website.

### 3.5.3 An Example: Windows XP

The Windows XP operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows XP provides support for multiple operating environments, or *subsystems*, with which application programs communicate via a message-passing mechanism. The application programs can be considered clients of the Windows XP subsystem server.

The message-passing facility in Windows XP is called the **local procedure-call (LPC)** facility. The LPC in Windows XP communicates between two processes on the same machine. It is similar to the standard RPC mechanism that is widely used, but it is optimized for and specific to Windows XP. Like Mach, Windows XP uses a port object to establish and maintain a connection between two processes. Every client that calls a subsystem needs a communication channel, which is provided by a port object and is never inherited. Windows XP uses two types of ports: connection ports and communication ports. They

are really the same but are given different names according to how they are used. Connection ports are named *objects* and are visible to all processes; they give applications a way to set up communication channels (Chapter 22). The communication works as follows:

- The client opens a handle to the subsystem's connection port object.
- The client sends a connection request.
- The server creates two private communication ports and returns the handle to one of them to the client.
- The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Windows XP uses two types of message-passing techniques over a port that the client specifies when it establishes the channel. The simplest, which is used for small messages, uses the port's message queue as intermediate storage and copies the message from one process to the other. Under this method, messages of up to 256 bytes can be sent.

If a client needs to send a larger message, it passes the message through a **section object**, which sets up a region of shared memory. The client has to decide when it sets up the channel whether or not it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created. Similarly, if the server decides that replies will be large, it creates a section object. So that the section object can be used, a small message is sent that contains a pointer and size information about the section object. This method is more complicated than the first method, but it avoids data copying. In both cases, a callback mechanism can be used when either the client or the server cannot respond immediately to a request. The callback mechanism allows them to perform asynchronous message handling. The structure of local procedure calls in Windows XP is shown in Figure 3.17.

It is important to note that the LPC facility in Windows XP is not part of the Win32 API and hence is not visible to the application programmer. Rather,

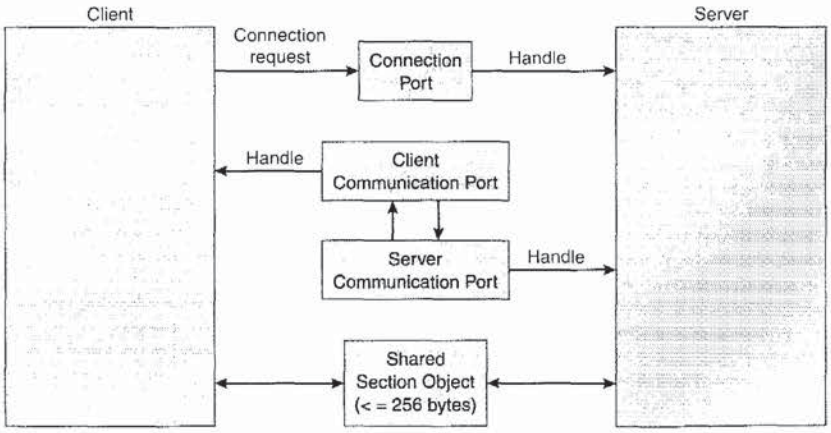



**Figure 3.17** Local procedure calls in Windows XP.

applications using the Win32 API invoke standard remote procedure calls. When the RPC is being invoked on a process on the same system, the RPC is indirectly handled through a local procedure call. LPCs are also used in a few other functions that are part of the Win32 API.

## 3.6 Communication in Client-Server Systems

In Section 3.4, we described how processes can communicate using shared memory and message passing. These techniques can be used for communication in client–server systems (1.12.2) as well. In this section, we explore three other strategies for communication in client–server systems: sockets, remote procedure calls (RPCs), and Java's remote method invocation (RMI).

### 3.6.1 Sockets

A **socket** is defined as an endpoint for communication. A pair of processes communicating over a network employ a pair of sockets—one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client–server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. Servers implementing specific services (such as telnet, ftp, and http) listen to well-known ports (a telnet server listens to port 23, an ftp server listens to port 21, and a web, or http, server listens to port 80). All ports below 1024 are considered *well known*; we can use them to implement standard services.

When a client process initiates a request for a connection, it is assigned a port by the host computer. This port is some arbitrary number greater than 1024. For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server. This situation is illustrated in Figure 3.18. The packets

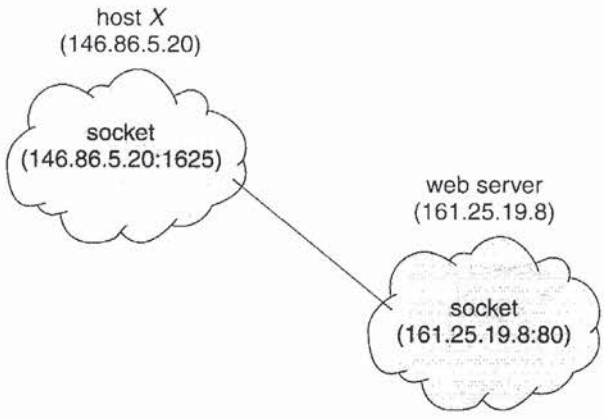



**Figure 3.18** Communication using sockets.

traveling between the hosts are delivered to the appropriate process based on the destination port number.

All connections must be unique. Therefore, if another process also on host X wished to establish another connection with the same web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures that all connections consist of a unique pair of sockets.

Although most program examples in this text use C, we will illustrate sockets using Java, as it provides a much easier interface to sockets and has a rich library for networking utilities. Those interested in socket programming in C or C++ should consult the bibliographical notes at the end of the chapter.

Java provides three different types of sockets. **Connection-oriented (TCP) sockets** are implemented with the `Socket` class. **Connectionless (UDP) sockets** use the `DatagramSocket` class. Finally, the `MulticastSocket` class is a subclass of the `DatagramSocket` class. A multicast socket allows data to be sent to multiple recipients.

Our example describes a date server that uses connection-oriented TCP sockets. The operation allows clients to request the current date and time from

```
import java.net.*;
import java.io.*;

public class DateServer
{
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);

      // now listen for connections
      while (true) {
        Socket client = sock.accept();

        PrintWriter pout = new
          PrintWriter(client.getOutputStream(), true);

        // write the Date to the socket
        pout.println(new java.util.Date().toString());

        // close the socket and resume
        // listening for connections
        client.close();
      }
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

**Figure 3.19** Date server.