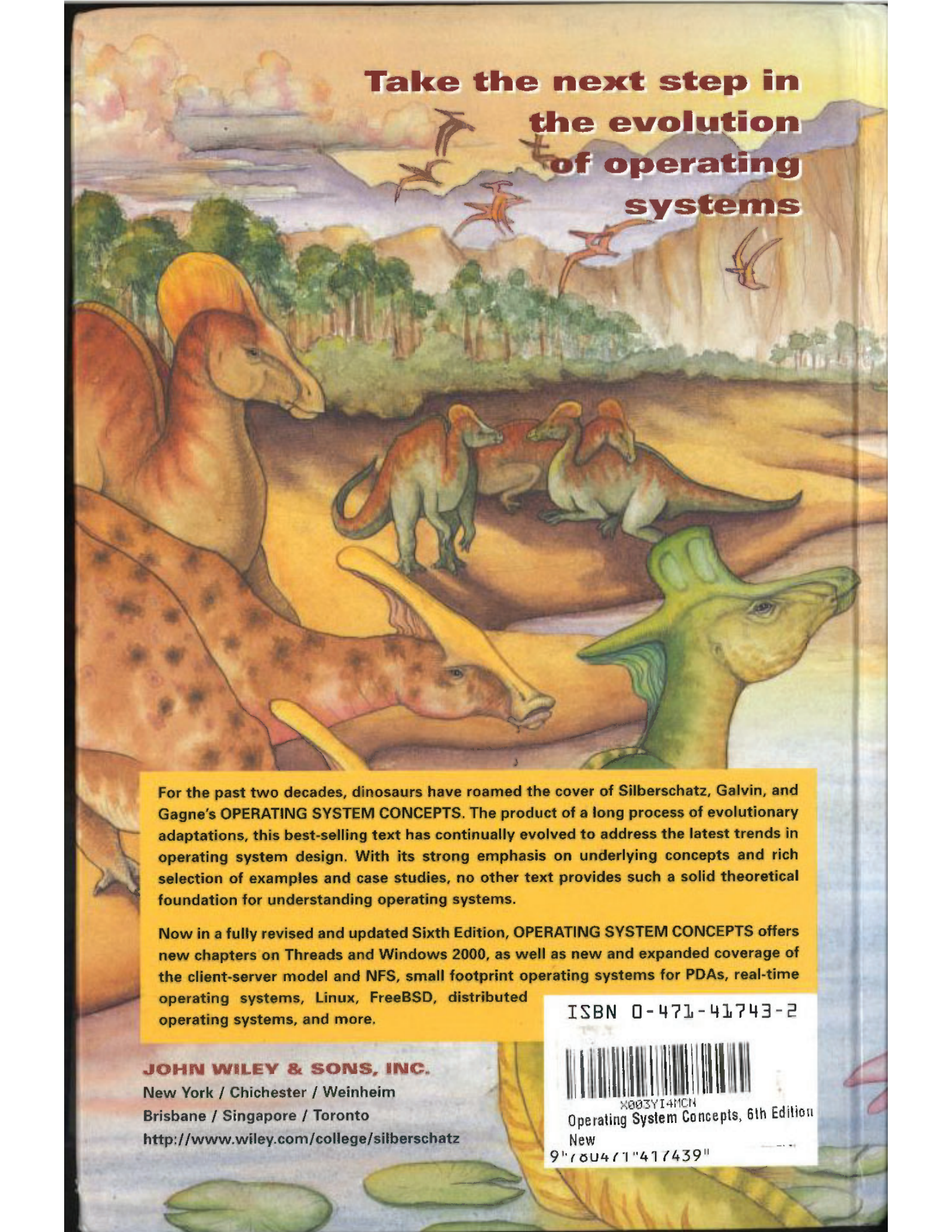


**SILBERSCHATZ
GALVIN
GAGNE**

**OPERATING
SYSTEM
CONCEPTS**

SIXTH EDITION



**Take the next step in
the evolution
of operating
systems**

For the past two decades, dinosaurs have roamed the cover of Silberschatz, Galvin, and Gagne's **OPERATING SYSTEM CONCEPTS**. The product of a long process of evolutionary adaptations, this best-selling text has continually evolved to address the latest trends in operating system design. With its strong emphasis on underlying concepts and rich selection of examples and case studies, no other text provides such a solid theoretical foundation for understanding operating systems.

Now in a fully revised and updated Sixth Edition, **OPERATING SYSTEM CONCEPTS** offers new chapters on Threads and Windows 2000, as well as new and expanded coverage of the client-server model and NFS, small footprint operating systems for PDAs, real-time operating systems, Linux, FreeBSD, distributed operating systems, and more.

JOHN WILEY & SONS, INC.

New York / Chichester / Weinheim

Brisbane / Singapore / Toronto

<http://www.wiley.com/college/silberschatz>

ISBN 0-471-41743-2



X883YI4MCH

Operating System Concepts, 6th Edition

New

9 780471 417439

ACQUISITIONS EDITOR	Paul Crockett
SENIOR MARKETING MANAGER	Katherine Hepburn
SENIOR PRODUCTION EDITOR	Ken Santor
COVER DESIGNER	Madelyn Lesure
COVER ART	Susan E. Cyr
SENIOR ILLUSTRATION COORDINATOR	Anna Melhorn

This book was set in Palatino by Abraham Silberschatz and printed and bound by Courier-Westford. The cover was printed by Phoenix Color Corporation.

This book is printed on acid-free paper.

The paper in this book was manufactured by a mill whose forest management programs include sustained yield harvesting of its timberlands. Sustained yield harvesting principles ensure that the numbers of trees cut each year does not exceed the amount of new growth.

Copyright © 2002 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, (508) 750-8400, fax (508) 750-4470. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, (212) 850-6011, fax (212) 850-6008, E-Mail: PERMREQ@WILEY.COM.

ISBN 0-471-41743-2

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

CONTENTS

PART ONE ■ OVERVIEW

Chapter 1 Introduction

- | | | | |
|----------------------------------|----|-----------------------------|----|
| 1.1 What Is an Operating System? | 3 | 1.8 Handheld Systems | 19 |
| 1.2 Mainframe Systems | 7 | 1.9 Feature Migration | 20 |
| 1.3 Desktop Systems | 11 | 1.10 Computing Environments | 21 |
| 1.4 Multiprocessor Systems | 12 | 1.11 Summary | 23 |
| 1.5 Distributed Systems | 14 | Exercises | 24 |
| 1.6 Clustered Systems | 16 | Bibliographical Notes | 25 |
| 1.7 Real-Time Systems | 17 | | |

Chapter 2 Computer-System Structures

- | | | | |
|-------------------------------|----|-----------------------|----|
| 2.1 Computer-System Operation | 27 | 2.6 Network Structure | 48 |
| 2.2 I/O Structure | 30 | 2.7 Summary | 51 |
| 2.3 Storage Structure | 34 | Exercises | 52 |
| 2.4 Storage Hierarchy | 38 | Bibliographical Notes | 54 |
| 2.5 Hardware Protection | 42 | | |

Chapter 3 Operating-System Structures

- 3.1 System Components 55
- 3.2 Operating-System Services 61
- 3.3 System Calls 63
- 3.4 System Programs 72
- 3.5 System Structure 74
- 3.6 Virtual Machines 80
- 3.7 System Design and Implementation 85
- 3.8 System Generation 88
- 3.9 Summary 89
 - Exercises 90
 - Bibliographical Notes 92

PART TWO ■ PROCESS MANAGEMENT

Chapter 4 Processes

- 4.1 Process Concept 95
- 4.2 Process Scheduling 99
- 4.3 Operations on Processes 103
- 4.4 Cooperating Processes 107
- 4.5 Interprocess Communication 109
- 4.6 Communication in Client-Server Systems 117
- 4.7 Summary 126
 - Exercises 127
 - Bibliographical Notes 128

Chapter 5 Threads

- 5.1 Overview 129
- 5.2 Multithreading Models 132
- 5.3 Threading Issues 135
- 5.4 Pthreads 139
- 5.5 Solaris 2 Threads 141
- 5.6 Window 2000 Threads 143
- 5.7 Linux Threads 144
- 5.8 Java Threads 145
- 5.9 Summary 147
 - Exercises 147
 - Bibliographical Notes 148

Chapter 6 CPU Scheduling

- 6.1 Basic Concepts 151
- 6.2 Scheduling Criteria 155
- 6.3 Scheduling Algorithms 157
- 6.4 Multiple-Processor Scheduling 169
- 6.5 Real-Time Scheduling 170
- 6.6 Algorithm Evaluation 172
- 6.7 Process Scheduling Models 177
- 6.8 Summary 184
 - Exercises 185
 - Bibliographical Notes 187

Chapter 7 Process Synchronization

- 7.1 Background 189
- 7.2 The Critical-Section Problem 191
- 7.3 Synchronization Hardware 197
- 7.4 Semaphores 201
- 7.5 Classic Problems of Synchronization 206
- 7.6 Critical Regions 211
- 7.7 Monitors 216
- 7.8 OS Synchronization 223
- 7.9 Atomic Transactions 225
- 7.10 Summary 235
 - Exercises 236
 - Bibliographical Notes 240

Chapter 8 Deadlocks

- 8.1 System Model 243
- 8.2 Deadlock Characterization 245
- 8.3 Methods for Handling Deadlocks 248
- 8.4 Deadlock Prevention 250
- 8.5 Deadlock Avoidance 253
- 8.6 Deadlock Detection 260
- 8.7 Recovery from Deadlock 264
- 8.8 Summary 266
 - Exercises 266
 - Bibliographical Notes 270

PART THREE ■ STORAGE MANAGEMENT

Chapter 9 Memory Management

- 9.1 Background 273
- 9.2 Swapping 280
- 9.3 Contiguous Memory Allocation 283
- 9.4 Paging 287
- 9.5 Segmentation 303
- 9.6 Segmentation with Paging 309
- 9.7 Summary 312
 - Exercises 313
 - Bibliographical Notes 316

Chapter 10 Virtual Memory

- 10.1 Background 317
- 10.2 Demand Paging 320
- 10.3 Process Creation 328
- 10.4 Page Replacement 330
- 10.5 Allocation of Frames 344
- 10.6 Thrashing 348
- 10.7 Operating-System Examples 353
- 10.8 Other Considerations 356
- 10.9 Summary 363
 - Exercises 364
 - Bibliographical Notes 369

Chapter 11 File-System Interface

- 11.1 File Concept 371
- 11.2 Access Methods 379
- 11.3 Directory Structure 383
- 11.4 File-System Mounting 393
- 11.5 File Sharing 395
- 11.6 Protection 402
- 11.7 Summary 406
- Exercises 407
- Bibliographical Notes 409

Chapter 12 File-System Implementation

- 12.1 File-System Structure 411
- 12.2 File-System Implementation 413
- 12.3 Directory Implementation 420
- 12.4 Allocation Methods 421
- 12.5 Free-Space Management 430
- 12.6 Efficiency and Performance 433
- 12.7 Recovery 437
- 12.8 Log-Structured File System 439
- 12.9 NFS 441
- 12.10 Summary 448
- Exercises 449
- Bibliographical Notes 451

PART FOUR ■ I/O SYSTEMS

Chapter 13 I/O Systems

- 13.1 Overview 455
- 13.2 I/O Hardware 456
- 13.3 Application I/O Interface 466
- 13.4 Kernel I/O Subsystem 472
- 13.5 Transforming I/O to Hardware Operations 478
- 13.6 STREAMS 481
- 13.7 Performance 483
- 13.8 Summary 487
- Exercises 487
- Bibliographical Notes 488

Chapter 14 Mass-Storage Structure

- 14.1 Disk Structure 491
- 14.2 Disk Scheduling 492
- 14.3 Disk Management 498
- 14.4 Swap-Space Management 502
- 14.5 RAID Structure 505
- 14.6 Disk Attachment 512
- 14.7 Stable-Storage Implementation 514
- 14.8 Tertiary-Storage Structure 516
- 14.9 Summary 526
- Exercises 528
- Bibliographical Notes 535

PART FIVE ■ DISTRIBUTED SYSTEMS

Chapter 15 Distributed System Structures

- | | | | |
|------------------------------|-----|-----------------------------|-----|
| 15.1 Background | 539 | 15.7 Design Issues | 564 |
| 15.2 Topology | 546 | 15.8 An Example: Networking | 566 |
| 15.3 Network Types | 548 | 15.9 Summary | 568 |
| 15.4 Communication | 551 | Exercises | 569 |
| 15.5 Communication Protocols | 558 | Bibliographical Notes | 571 |
| 15.6 Robustness | 562 | | |

Chapter 16 Distributed File Systems

- | | | | |
|--|-----|-----------------------|-----|
| 16.1 Background | 573 | 16.6 An Example: AFS | 586 |
| 16.2 Naming and Transparency | 575 | 16.7 Summary | 591 |
| 16.3 Remote File Access | 579 | Exercises | 592 |
| 16.4 Stateful Versus Stateless Service | 583 | Bibliographical Notes | 593 |
| 16.5 File Replication | 585 | | |

Chapter 17 Distributed Coordination

- | | | | |
|--------------------------|-----|--------------------------|-----|
| 17.1 Event Ordering | 595 | 17.6 Election Algorithms | 618 |
| 17.2 Mutual Exclusion | 598 | 17.7 Reaching Agreement | 620 |
| 17.3 Atomicity | 601 | 17.8 Summary | 623 |
| 17.4 Concurrency Control | 605 | Exercises | 624 |
| 17.5 Deadlock Handling | 610 | Bibliographical Notes | 625 |

PART SIX ■ PROTECTION AND SECURITY

Chapter 18 Protection

- | | | | |
|--------------------------------------|-----|--------------------------------|-----|
| 18.1 Goals of Protection | 629 | 18.6 Capability-Based Systems | 645 |
| 18.2 Domain of Protection | 630 | 18.7 Language-Based Protection | 648 |
| 18.3 Access Matrix | 636 | 18.8 Summary | 654 |
| 18.4 Implementation of Access Matrix | 640 | Exercises | 655 |
| 18.5 Revocation of Access Rights | 643 | Bibliographical Notes | 656 |

Chapter 19 Security

- 19.1 The Security Problem 657
- 19.2 User Authentication 659
- 19.3 Program Threats 663
- 19.4 System Threats 666
- 19.5 Securing Systems and Facilities 671
- 19.6 Intrusion Detection 674
- 19.7 Cryptography 680
- 19.8 Computer-Security Classifications 686
- 19.9 An Example: Windows NT 687
- 19.10 Summary 689
 - Exercises 690
 - Bibliographical Notes 691

PART SEVEN ■ CASE STUDIES**Chapter 20 The Linux System**

- 20.1 History 695
- 20.2 Design Principles 700
- 20.3 Kernel Modules 703
- 20.4 Process Management 707
- 20.5 Scheduling 711
- 20.6 Memory Management 716
- 20.7 File Systems 724
- 20.8 Input and Output 729
- 20.9 Interprocess Communication 732
- 20.10 Network Structure 734
- 20.11 Security 737
- 20.12 Summary 739
 - Exercises 740
 - Bibliographical Notes 741

Chapter 21 Windows 2000

- 21.1 History 743
- 21.2 Design Principles 744
- 21.3 System Components 746
- 21.4 Environmental Subsystems 763
- 21.5 File System 766
- 21.6 Networking 774
- 21.7 Programmer Interface 780
- 21.8 Summary 787
 - Exercises 787
 - Bibliographical Notes 788

Chapter 22 Historical Perspective

- 22.1 Early Systems 789
- 22.2 Atlas 796
- 22.3 XDS-940 797
- 22.4 THE 798
- 22.5 RC 4000 799
- 22.6 CTSS 800
- 22.7 MULTICS 800
- 22.8 OS/360 801
- 22.9 Mach 803
- 22.10 Other Systems 804

Appendix A The FreeBSD System (contents online)

- | | | | |
|--------------------------|------|--------------------------------|------|
| A.1 History | A807 | A.7 File System | A834 |
| A.2 Design Principles | A813 | A.8 I/O System | A842 |
| A.3 Programmer Interface | A815 | A.9 Interprocess Communication | A846 |
| A.4 User Interface | A823 | A.10 Summary | A852 |
| A.5 Process Management | A827 | Exercises | A852 |
| A.6 Memory Management | A831 | Bibliographical Notes | A853 |

Appendix B The Mach System (contents online)

- | | | | |
|--------------------------------|------|--------------------------|------|
| B.1 History | A855 | B.7 Programmer Interface | A880 |
| B.2 Design Principles | A857 | B.8 Summary | A881 |
| B.3 System Components | A858 | Exercises | A882 |
| B.4 Process Management | A862 | Bibliographical Notes | A883 |
| B.5 Interprocess Communication | A868 | Credits | A885 |
| B.6 Memory Management | A874 | | |

Appendix C The Nachos System (contents online)

- | | | | |
|--------------------------------|------|-----------------------|------|
| C.1 Overview | A888 | C.5 Conclusions | A900 |
| C.2 Nachos Software Structure | A890 | Bibliographical Notes | A901 |
| C.3 Sample Assignments | A893 | Credits | A902 |
| C.4 Obtaining a Copy of Nachos | A898 | | |

Bibliography 807

Credits 837

Index 839

Chapter 1



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 a user of a computer 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. Handheld computer operating systems 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.

To understand what operating systems are, we must first understand how they have developed. In this chapter, we trace the development of operating systems from the first hands-on systems through multiprogrammed and time-shared systems to PCs, and handheld computers. We also discuss operating system variations, such as parallel, real-time, and embedded systems. As we move through the various stages, we see how the components of operating systems evolved as natural solutions to problems in early computer systems.

1.1 ■ What Is an Operating System?

An operating system is an important part of almost every 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).

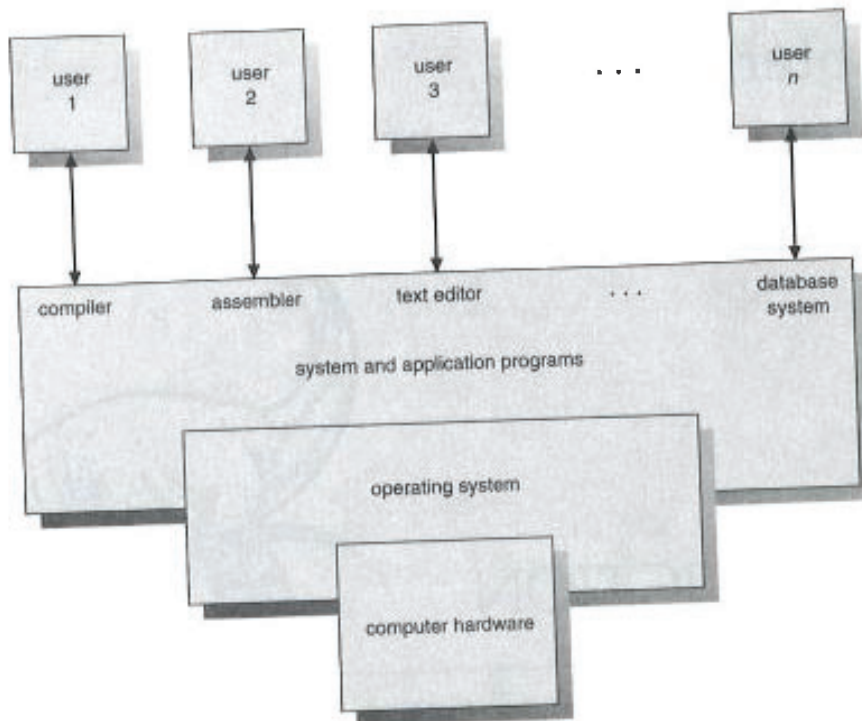


Figure 1.1 Abstract view of the components of a computer system.

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

The components of a computer system are its hardware, software, and data. The operating system provides the means for the 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. Operating systems can be explored from two viewpoints: the user and the system.

1.1.1 User View

The user view of the computer varies by 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, 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. Performance is important to the user, but it does not matter if most of the system is sitting idle, waiting for the slow I/O speed of the user.

Some users sit 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 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.

Other 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. These devices are mostly standalone, used singly by individual users. Some are connected to networks, either directly by wire or (more often) through wireless modems. Due to power and interface limitations they perform relatively few remote operations. The 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 a numeric keypad, and may turn indicator lights on or off to show status, but mostly they and their operating systems are designed to run without user intervention.

1.1.2 System View

From the computer's point of view, the operating system is the program that is most intimate with the hardware. We can view an operating system as a **resource allocator**. A computer system has many resources—hardware and software—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.

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.

In general, however, we have no completely adequate definition of an operating system. Operating systems exist because they are a reasonable way to solve the problem of creating a usable computing system. The fundamental

Chapter 13

I/O SYSTEMS



The two main jobs of a computer are I/O and processing. In many cases, the main job is I/O and the processing is merely incidental. For instance, when we browse a web page or edit a file, our immediate interest is to read or enter some information, not to compute an answer.

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. Although related topics appear in other chapters, here we bring together the pieces to paint a complete picture of I/O. First, we describe the basics of I/O hardware, because the nature of the hardware interface places requirements on the internal facilities of the operating system. Next, we discuss the I/O services provided by the operating system, and the embodiment of these services in the application I/O interface. Then, we explain how the operating system bridges the gap between the hardware interface and the application interface. We also discuss the UNIX System V STREAMS mechanism, which enables an application to assemble pipelines of driver code dynamically. Finally, we discuss the performance aspects of I/O, and the principles of operating-system design that improve the I/O performance.

13.1 ■ Overview

The control of devices connected to the computer is a major concern of operating-system designers. Because I/O devices vary so widely in their function and speed (consider a mouse, a hard disk, and a CD-ROM jukebox), a

variety of methods are needed to control them. These methods form the *I/O subsystem* of the kernel, which separates the rest of the kernel from the complexity of managing I/O devices.

I/O-device technology exhibits two conflicting trends. On one hand, we see increasing standardization of software and hardware interfaces. This trend helps us to incorporate improved device generations into existing computers and operating systems. On the other hand, we see an increasingly broad variety of I/O devices. Some new devices are so unlike previous devices that it is a challenge to incorporate them into our computers and operating systems. This challenge is met by a combination of hardware and software techniques. The basic I/O hardware elements, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices. To encapsulate the details and oddities of different devices, the kernel of an operating system is structured to use device-driver modules. The device drivers present a uniform device-access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

13.2 ■ I/O Hardware

Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks, tapes), transmission devices (network cards, modems), and human-interface devices (screen, keyboard, mouse). Other devices are more specialized, such as the steering of a military fighter jet or a space shuttle. In these aircraft, a human gives input to the flight computer via a joystick, and the computer sends output commands that cause motors to move rudders, flaps, and thrusters.

Despite the incredible variety of I/O devices, we need only a few concepts to understand how the devices are attached, and how the software can control the hardware.

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point (or port), for example, a serial port. If one or more devices use a common set of wires, the connection is called a *bus*. A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings. When device A has a cable that plugs into device B, and device B has a cable that plugs into device C, and device C plugs into a port on the computer, this arrangement is called a **daisy chain**. A daisy chain usually operates as a bus.

Buses are used widely in computer architecture. Figure 13.1 shows a typical PC bus structure. This figure shows a **PCI bus** (the common PC system bus) that connects the processor-memory subsystem to the fast devices, and an **expansion bus** that connects relatively slow devices such as the keyboard and

raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected. The interrupt mechanism is also used to handle a wide variety of exceptions, such as dividing by zero, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode. The events that trigger interrupts have a common property: They are occurrences that induce the CPU to execute an urgent, self-contained routine.

An operating system has other good uses for an efficient hardware mechanism that saves a small amount of processor state, and then calls a privileged routine in the kernel. For example, many operating systems use the interrupt mechanism for virtual-memory paging. A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel. This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt.

Another example is found in the implementation of system calls. A *system call* is a function called by an application to invoke a kernel service. The system call checks the arguments given by the application, builds a data structure to convey the arguments to the kernel, and then executes a special instruction called a **software interrupt** (or a **trap**). This instruction has an operand that identifies the desired kernel service. When the system call executes the trap instruction, the interrupt hardware saves the state of the user code, switches to supervisor mode, and dispatches to the kernel routine that implements the requested service. The trap is given a relatively low interrupt priority compared to those assigned to device interrupts—executing a system call on behalf of an application is less urgent than servicing a device controller before its FIFO queue overflows and loses data.

Interrupts can also be used to manage the flow of control within the kernel. For example, consider the processing required to complete a disk read. One step is to copy data from kernel space to the user buffer. This copying is time consuming but not urgent—it should not block other high-priority interrupt handling. Another step is to start the next pending I/O for that disk drive. This step has higher priority: If the disks are to be used efficiently, we need to start the next I/O as soon as the previous one completes. Consequently, a *pair* of interrupt handlers implements the kernel code that completes a disk read. The high-priority handler records the I/O status, clears the device interrupt, starts the next pending I/O, and raises a low-priority interrupt to complete the work. Later, when the CPU is not occupied with high-priority work, the low-priority interrupt will be dispatched. The corresponding handler completes the user-level I/O by copying data from kernel buffers to the application space, and then by calling the scheduler to place the application on the ready queue.

A threaded kernel architecture is well suited to implement multiple interrupt priorities and to enforce the precedence of interrupt handling over back-

access style is also good for output devices such as printers or audio boards, which naturally fit the concept of a linear stream of bytes.

13.3.2 Network Devices

Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the `read()`-`write()`-`seek()` interface used for disks. One interface available in many operating systems, including UNIX and Windows NT, is the network **socket** interface.

Think of a wall socket for electricity: Any electrical appliance can be plugged in. By analogy, the system calls in the socket interface enable an application to create a socket, to connect a local socket to a remote address (which plugs this application into a socket created by another application), to listen for any remote application to plug into the local socket, and to send and receive packets over the connection. To support the implementation of servers, the socket interface also provides a function called `select()` that manages a set of sockets. A call to `select()` returns information about which sockets have a packet waiting to be received, and which sockets have room to accept a packet to be sent. The use of `select()` eliminates the polling and busy waiting that would otherwise be necessary for network I/O. These functions encapsulate the essential behaviors of networks, greatly facilitating the creation of distributed applications that can use any underlying network hardware and protocol stack.

Many other approaches to interprocess communication and network communication have been implemented. For instance, Windows NT provides one interface to the network interface card, and a second interface to the network protocols (Section 21.6). In UNIX, which has a long history as a proving ground for network technology, we find half-duplex pipes, full-duplex FIFOs, full-duplex STREAMS, message queues, and sockets. Information on UNIX networking is given in Section A.9.

13.3.3 Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:

- Give the current time
- Give the elapsed time
- Set a timer to trigger operation X at time T

These functions are used heavily by the operating system, and also by time-sensitive applications. Unfortunately, the system calls that implement these functions are not standardized across operating systems.

The hardware to measure elapsed time and to trigger operations is called a **programmable interval timer**. It can be set to wait a certain amount of time and then to generate an interrupt. It can be set to do this operation once, or to repeat the process, to generate periodic interrupts. The scheduler uses this mechanism to generate an interrupt that will preempt a process at the end of its time slice. The disk I/O subsystem uses it to invoke the flushing of dirty cache buffers to disk periodically, and the network subsystem uses it to cancel operations that are proceeding too slowly because of network congestion or failures. The operating system may also provide an interface for user processes to use timers. The operating system can support more timer requests than the number of timer hardware channels by simulating virtual clocks. To do so, the kernel (or the timer device driver) maintains a list of interrupts wanted by its own routines and by user requests, sorted in earliest-time-first order. It sets the timer for the earliest time. When the timer interrupts, the kernel signals the requester, and reloads the timer with the next earliest time.

On many computers, the interrupt rate generated by the ticking of the hardware clock is between 18 and 60 ticks per second. This resolution is coarse, since a modern computer can execute hundreds of millions of instructions per second. The precision of triggers is limited by the coarse resolution of the timer, together with the overhead of maintaining virtual clocks. And, if the timer ticks are used to maintain the system time-of-day clock, the system clock can drift. In most computers, the hardware clock is constructed from a high-frequency counter. In some computers, the value of this counter can be read from a device register, in which case the counter can be considered to be a high-resolution clock. Although this clock does not generate interrupts, it offers accurate measurements of time intervals.

13.3.4 Blocking and Nonblocking I/O

Another aspect of the system-call interface relates to the choice between blocking I/O and nonblocking (or asynchronous) I/O. When an application issues a **blocking** system call, the execution of the application is suspended. The application is moved from the operating system's run queue to a wait queue. After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution, at which time it will receive the values returned by the system call. The physical actions performed by I/O devices are generally asynchronous—they take a varying or unpredictable amount of time. Nevertheless, most operating systems use blocking system calls for the application interface, because blocking application code is easier to understand than nonblocking application code.

Some user-level processes need **nonblocking** I/O. One example is a user interface that receives keyboard and mouse input while processing and displaying data on the screen. Another example is a video application that reads

frames from a file on disk while simultaneously decompressing and displaying the output on the display.

One way that an application writer can overlap execution with I/O is to write a multithreaded application. Some threads can perform blocking system calls, while others continue executing. The Solaris developers used this technique to implement a user-level library for asynchronous I/O, freeing the application writer from that task. Some operating systems provide nonblocking I/O system calls. A nonblocking call does not halt the execution of the application for an extended time. Instead, it returns quickly, with a return value that indicates how many bytes were transferred.

An alternative to a nonblocking system call is an asynchronous system call. An asynchronous call returns immediately, without waiting for the I/O to complete. The application continues to execute its code. The completion of the I/O at some future time is communicated to the application, either through the setting of some variable in the address space of the application, or through the triggering of a signal or software interrupt or a call-back routine that is executed outside the linear control flow of the application. The difference between nonblocking and asynchronous system calls is that a nonblocking `read()` returns immediately with whatever data are available—the full number of bytes requested, fewer, or none at all. An asynchronous `read()` call requests a transfer that will be performed in its entirety, but that will complete at some future time.

A good example of nonblocking behavior is the `select()` system call for network sockets. This system call takes an argument that specifies a maximum waiting time. By setting it to 0, an application can poll for network activity without blocking. But using `select()` introduces extra overhead, because the `select()` call only checks whether I/O is possible. For a data transfer, `select()` must be followed by some kind of `read()` or `write()` command. A variation of this approach, found in Mach, is a blocking multiple-read call. It specifies desired reads for several devices in one system call, and returns as soon as any one of them completes.

13.4 ■ Kernel I/O Subsystem

Kernels provide many services related to I/O. Several services—scheduling, buffering, caching, spooling, device reservation, and error handling—are provided by the kernel's I/O subsystem and build on the hardware and device-driver infrastructure.

13.4.1 I/O Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share

device access fairly among processes, and can reduce the average waiting time for I/O to complete. Here is a simple example to illustrate the opportunity. Suppose that a disk arm is near the beginning of a disk, and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one near the beginning, and application 3 requests one in the middle of the disk. The operating system can reduce the distance that the disk arm travels by serving the applications in order 2, 3, 1. Rearranging the order of service in this way is the essence of I/O scheduling.

Operating-system developers implement scheduling by maintaining a queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. The operating system may also try to be fair, so that no one application receives especially poor service, or it may give priority service for delay-sensitive requests. For instance, requests from the virtual-memory subsystem may take priority over application requests. Several scheduling algorithms for disk I/O are detailed in Section 14.2.

One way that the I/O subsystem improves the efficiency of the computer is by scheduling I/O operations. Another way is by using storage space in main memory or on disk, via techniques called buffering, caching, and spooling.

13.4.2 Buffering

A **buffer** is a memory area that stores data while they are transferred between two devices or between a device and an application. Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. Suppose, for example, that a file is being received via modem for storage on the hard disk. The modem is about a thousand times slower than the hard disk. So a buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer of data has arrived, the buffer can be written to disk in a single operation. Since the disk write is not instantaneous and the modem still needs a place to store additional incoming data, two buffers are used. After the modem fills the first buffer, the disk write is requested. The modem then starts to fill the second buffer while the first buffer is written to disk. By the time the modem has filled the second buffer, the disk write from the first one should have completed, so the modem can switch back to the first buffer while the disk writes the second one. This **double buffering** decouples the producer of data from the consumer, thus relaxing timing requirements between them. The need for this decoupling is illustrated in Figure 13.8, which lists the enormous differences in device speeds for typical computer hardware.

A second use of buffering is to adapt between devices that have different data-transfer sizes. Such disparities are especially common in computer

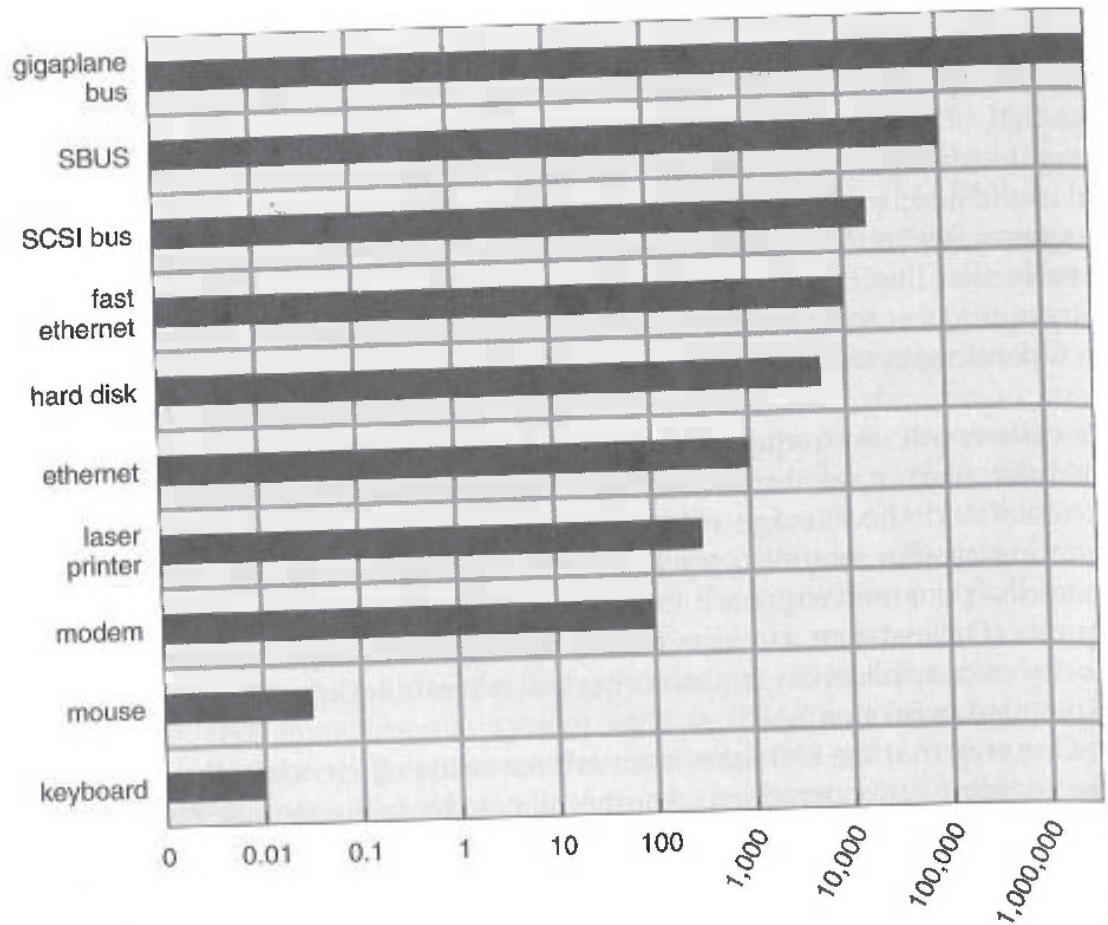


Figure 13.8 Sun Enterprise 6000 device-transfer rates (logarithmic).

networking, where buffers are used widely for fragmentation and reassembly of messages. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.

A third use of buffering is to support copy semantics for application I/O. An example will clarify the meaning of "copy semantics." Suppose that an application has a buffer of data that it wishes to write to disk. It calls the `write()` system call, providing a pointer to the buffer and an integer specifying the number of bytes to write. After the system call returns, what happens if the application changes the contents of the buffer? With **copy semantics**, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application's buffer. A simple way that the operating system can guarantee copy semantics is for the `write()` system call to copy the application data into a kernel buffer before returning control to the application. The disk write is performed from the kernel buffer, so that subsequent changes to the application buffer have no effect. Copying of data between kernel buffers and application

data space is common in operating systems, despite the overhead that this operation introduces, because of the clean semantics. The same effect can be obtained more efficiently by clever use of virtual-memory mapping and copy-on-write page protection.

13.4.3 Caching

A cache is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches. The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, just holds a copy on faster storage of an item that resides elsewhere.

Caching and buffering are distinct functions, but sometimes a region of memory can be used for both purposes. For instance, to preserve copy semantics and to enable efficient scheduling of disk I/O, the operating system uses buffers in main memory to hold disk data. These buffers are also used as a cache, to improve the I/O efficiency for files that are shared by applications or that are being written and reread rapidly. When the kernel receives a file I/O request, the kernel first accesses the buffer cache to see whether that region of the file is already available in main memory. If so, a physical disk I/O can be avoided or deferred. Also, disk writes are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules. This strategy of delaying writes to improve I/O efficiency is discussed, in the context of remote file access, in Section 16.3.

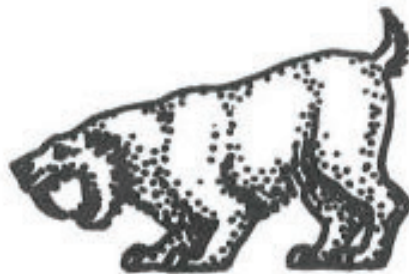
13.4.4 Spooling and Device Reservation

A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together. The operating system solves this problem by intercepting all output to the printer. Each application's output is spooled to a separate disk file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer. The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In other operating systems, it is handled by an in-kernel thread. In either case, the operating system provides a control interface that enables users and system administrators to display the queue, to remove unwanted jobs before those jobs print, to suspend printing while the printer is serviced, and so on.

Some devices, such as tape drives and printers, cannot usefully multiplex the I/O requests of multiple concurrent applications. Spooling is one way that

Chapter 20

THE LINUX SYSTEM



Appendix A discusses the internal workings of the 4.3BSD operating system. BSD is just one of the UNIX-like systems. Linux is another UNIX-like system that has gained popularity in recent years. In this chapter, we look at the history and development of Linux, and cover the user and programmer interfaces that Linux presents—interfaces that owe a great deal to the UNIX tradition. We also discuss the internal methods by which Linux implements these interfaces. However, since Linux has been designed to run as many standard UNIX applications as possible, it has much in common with existing UNIX implementations. We do not duplicate the basic description of UNIX given in Appendix A.

Linux is a rapidly evolving operating system. This chapter describes specifically the Linux 2.2 kernel, which was released in January, 1999.

20.1 ■ History

Linux looks and feels much like any other UNIX system; indeed, UNIX compatibility has been a major design goal of the Linux project. However, Linux is much younger than most UNIX systems. Its development began in 1991, when a Finnish student, Linus Torvalds, wrote and christened Linux, a small but self-contained kernel for the 80386 processor, the first true 32-bit processor in Intel's range of PC-compatible CPUs.

Early in its development, the Linux source code was made available for free on the Internet. As a result, Linux's history has been one of collaboration by many users from all around the world, corresponding almost exclusively over

the Internet. From an initial kernel that partially implemented a small subset of the UNIX system services, the Linux system has grown to include much UNIX functionality.

In its early days, Linux development revolved largely around the central operating-system kernel—the core, privileged executive that manages all system resources and that interacts directly with the computer hardware. We need much more than this kernel to produce a full operating system, of course. It is useful to make the distinction between the Linux kernel and a Linux system. The Linux kernel is an entirely original piece of software developed from scratch by the Linux community. The Linux system, as we know it today, includes a multitude of components, some written from scratch, others borrowed from other development projects, and others created in collaboration with other teams.

The basic Linux system is a standard environment for applications and user programming, but it does not enforce any standard means of managing the available functionality as a whole. As Linux has matured, there has been a need for another layer of functionality on top of the Linux system. A Linux **distribution** includes all the standard components of the Linux system, plus a set of administrative tools to simplify the initial installation and subsequent upgrading of Linux, and to manage installation and deinstallation of other packages on the system. A modern distribution also typically includes tools for management of file systems, creation and management of user accounts, administration of networks, and so on.

20.1.1 The Linux Kernel

The first Linux kernel released to the public was Version 0.01, dated May 14, 1991. It had no networking, ran on only 80386-compatible Intel processors and PC hardware, and had extremely limited device-driver support. The virtual-memory subsystem was also fairly basic and included no support for memory-mapped files; however, even this early incarnation supported shared pages with copy-on-write. The only file system supported was the Minix file system—the first Linux kernels were cross-developed on a Minix platform. However, the kernel did implement proper UNIX processes with protected address spaces.

The next milestone version, Linux 1.0, was released on March 14, 1994. This release culminated three years of rapid development of the Linux kernel. Perhaps the single biggest new feature was networking: 1.0 included support for UNIX's standard TCP/IP networking protocols, as well as a BSD-compatible socket interface for networking programming. Device-driver support was added for running IP over an Ethernet or (using PPP or SLIP protocols) over serial lines or modems.

The 1.0 kernel also included a new, much enhanced file system without the limitations of the original Minix file system, and supported a range of SCSI controllers for high-performance disk access. The developers extended the

virtual-memory subsystem to support paging to swap files and memory mapping of arbitrary files (but only read-only memory mapping was implemented in 1.0).

A range of extra hardware support was also included in this release. Although still restricted to the Intel PC platform, hardware support had grown to include floppy-disk and CD-ROM devices, as well as sound cards, a range of mice, and international keyboards. Floating-point emulation was also provided in the kernel for 80386 users who had no 80387 math coprocessor, and System V UNIX-style **interprocess communication (IPC)**, including shared memory, semaphores, and message queues, was implemented. Simple support for dynamically loadable and unloadable kernel modules was also provided.

At this point, development started on the 1.1 kernel stream, but numerous bug-fix patches were released subsequently against 1.0. This pattern was adopted as the standard numbering convention for Linux kernels: Kernels with an odd minor-version number such as 1.1, 1.3, or 2.1 are **development kernels**; even-numbered minor-version numbers are stable, **production kernels**. Updates against the stable kernels are intended as only remedial versions, whereas the development kernels may include newer and relatively untested functionality.

In March, 1995, the 1.2 kernel was released. This release did not offer nearly the same improvement in functionality as the 1.0 release, but it did include support for a much wider variety of hardware, including the new PCI hardware bus architecture. Developers added another PC-specific feature—support for the 80386 CPU's virtual 8086 mode—to allow emulation of the DOS operating system for PC computers. They updated the networking stack to provide support for the IPX protocol, and made the IP implementation more complete by including accounting and firewalling functionality.

The 1.2 kernel also was the final PC-only Linux kernel. The source distribution for Linux 1.2 included partially implemented support for SPARC, Alpha, and MIPS CPUs, but full integration of these other architectures did not begin until after the 1.2 stable kernel was released.

The Linux 1.2 release concentrated on wider hardware support and more complete implementations of existing functionality. Much new functionality was under development at the time, but integration of the new code into the main kernel source code had been deferred until after the stable 1.2 kernel had been released. As a result, the 1.3 development stream saw a great deal of new functionality added to the kernel.

This work was finally released as Linux 2.0 in June, 1996. This release was given a major version-number increment on account of two major new capabilities: support for multiple architectures, including a fully 64-bit native Alpha port, and support for multiprocessor architectures. Linux distributions based on 2.0 are also available for the Motorola 68000-series processors and for Sun's SPARC systems. A derived version of Linux running on top of the Mach Microkernel also runs on PC and PowerMac systems.

Because of the modular structure, additional environmental subsystems can be added without affecting the executive. In addition, Windows 2000 uses loadable drivers in the I/O system, so new file systems, new kinds of I/O devices, and new kinds of networking can be added while the system is running. Windows 2000 uses a client-server model like the Mach operating system, and supports distributed processing by remote procedure calls (RPCs) as defined by the Open Software Foundation.

An operating system is **portable** if it can be moved from one hardware architecture to another with relatively few changes. Windows 2000 is designed to be portable. As is true of the UNIX operating system, the majority of the system is written in C and C++. All processor-dependent code is isolated in a dynamic link library (DLL) called the **hardware-abstraction layer (HAL)**. A DLL is a file that gets mapped into a process' address space such that any functions in the DLL appear to be part of the process. The upper layers of Windows 2000 depend on HAL, rather than on the underlying hardware, and that helps Windows 2000 to be portable. HAL manipulates hardware directly, isolating the rest of Windows 2000 from hardware differences among the platforms on which it runs.

Reliability is the ability to handle error conditions, including the ability of the operating system to protect itself and its users from defective or malicious software. Windows 2000 resists defects and attacks by using hardware protection for virtual memory, and software protection mechanisms for operating-system resources. Also, Windows 2000 comes with a native file system—the **NTFS file system**—that recovers automatically from many kinds of file-system errors after a system crash. Windows NT Version 4.0 has a C-2 security classification from the U.S. government, which signifies a moderate level of protection from defective software and malicious attacks. Windows 2000 is currently under evaluation by the government for that classification as well. For more information about security classifications, see Section 19.8.

Windows 2000 provides source-level **compatibility** to applications that follow the IEEE 1003.1 (POSIX) standard. Thus, they can be compiled to run on Windows 2000 without changes to the source code. In addition, Windows 2000 can run the executable binaries for many programs compiled for Intel X86 architectures running MS-DOS, 16-bit Windows, OS/2, LAN Manager, and 32-bit Windows, by using the environmental subsystems mentioned earlier. These environmental subsystems support a variety of file systems, including the MS-DOS FAT file system, the OS/2 HPFS file system, the ISO9660 CD file system, and NTFS. Windows 2000' binary compatibility, however, is not perfect. In MS-DOS, for example, applications can access hardware ports directly. For reliability and security, Windows 2000 prohibits such access.

Windows 2000 is designed to afford good **performance**. The subsystems that constitute Windows 2000 can communicate with one another efficiently by a **local-procedure-call (LPC)** facility that provides high-performance message passing. Except for the kernel, threads in the subsystems of Windows 2000 can