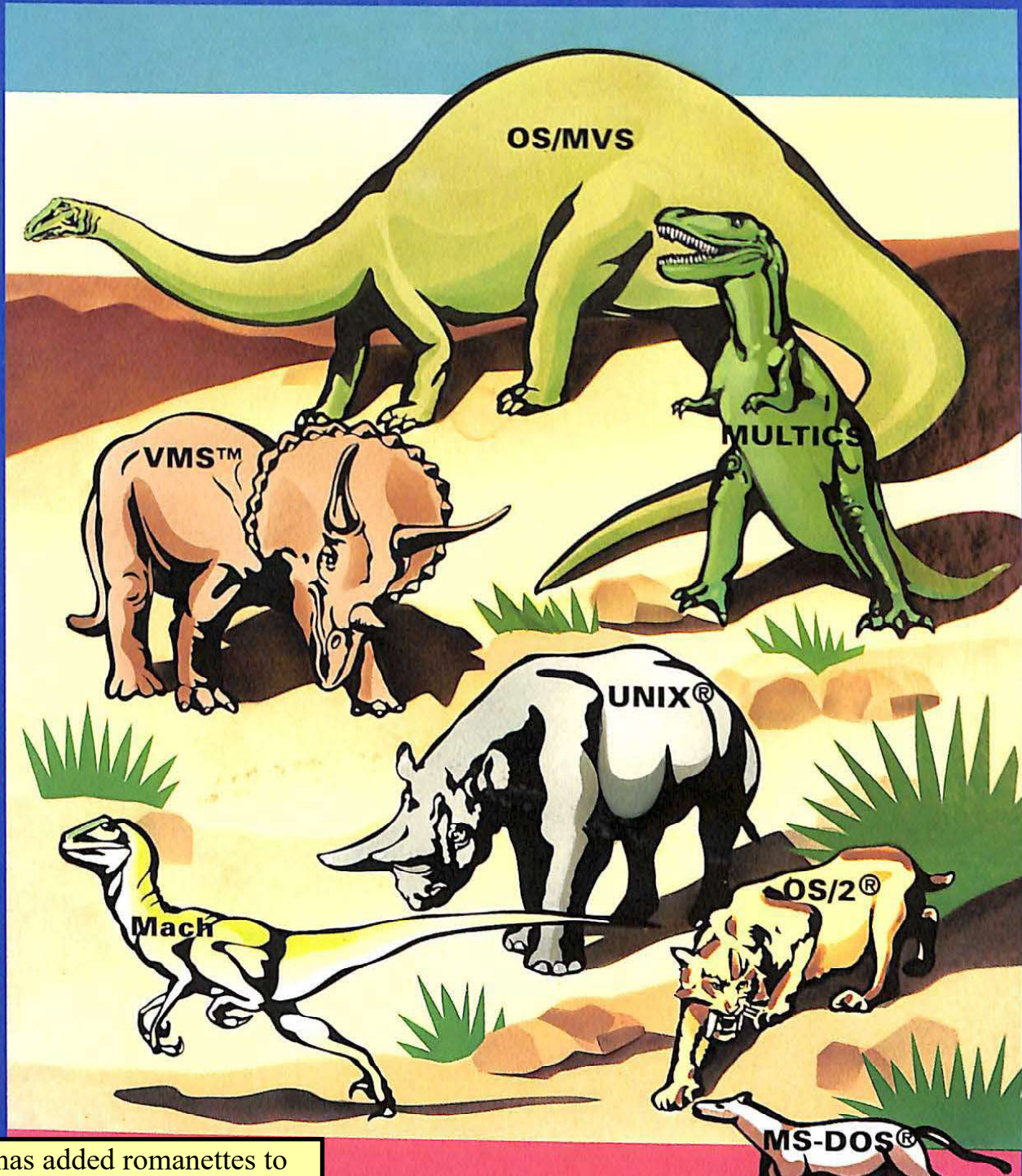


A. Silberschatz J. Peterson P. Galvin

Operating System Concepts

THIRD EDITION



Petitioner has added romanettes to pages (i)-(iv). Otherwise, it leaves the original page numbering.

DROPBOX, INC., ET AL.
EXHIBIT 1015

(i)

Operating System Concepts THIRD EDITION

Abraham Silberschatz University of Texas at Austin

James L. Peterson IBM Corporation

Peter B. Galvin Brown University



Addison-Wesley Publishing Company

Reading, Massachusetts ■ Menlo Park, California ■ New York
Don Mills, Ontario ■ Wokingham, England ■ Amsterdam ■ Bonn
Sydney ■ Singapore ■ Tokyo ■ Madrid ■ San Juan ■ Milan ■ Paris

This book is in the Addison-Wesley Series in Computer Science.
Consulting Editor: Michael Harrison

VMS is a trademark of Digital Equipment Corporation. UNIX is a registered trademark of AT&T Bell Laboratories. OS/2 is a registered trademark of International Business Machines Corporation. MS/DOS is a registered trademark of Microsoft Corporation. NFS is a registered trademark of Sun Microsystems, Inc. Macintosh is a registered trademark licensed to Apple Computer, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Fig. 3.8 from Iaccobucci, *OS/2 Programmer's Guide*, © 1988, McGraw-Hill, Inc., New York, New York. Fig. 1.7, p. 20. Reprinted with permission of the publisher.

Fig. 8.15 reprinted with permission from *IBM Systems Journal*, Vol. 10, No. 3, © 1971 International Business Machines Corporation.

Fig. 15.1 from Leffler/McKusick/Karels/Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, © 1989, Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Fig. 1.1. Reprinted with permission of the publisher.

Figs. 16.1, 16.5, and 16.6 reproduced with permission from Open Software Foundation, Inc. Excerpted from Mach Lecture Series, OSF, October 1989, Cambridge, Massachusetts.

Figs. 16.1 and 16.6 presented by R. Rashid of Carnegie-Mellon University and Fig. 16.5 presented by D. Julin of Carnegie-Mellon University.

Fig. 16.4 from Accetta/Baron/Bolosky/Golub/Rashid/Tevanian/Young, "Mach: a new kernel foundation for UNIX development," *Proceedings of Summer USENIX*, June 1986, Atlanta, Georgia. Reprinted with permission of the authors.

Library of Congress Cataloging-in-Publication Data

Silberschatz, Abraham.

Operating system concepts / Abraham Silberschatz, James L. Peterson, and Peter Galvin. -- 3rd ed.

p. cm.

Includes bibliographical references and index.

ISBN 0-201-51379-X

1. Operating systems (Computers) I. Peterson, James Lyle.

II. Galvin, Peter. III. Title.

QA76.76.063S5583 1991

005.4'3--dc20

90-44561
CIP

Reprinted with corrections April, 1992

Copyright © 1991 by Addison-Wesley Publishing Company, 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, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

5 6 7 8 9 10 DO 95949392

*To my parents, Wira and Mietek,
my wife, Haya,
and my children, Lemor, Steven and Aaron.*

Avi Silberschatz

*To my wife, Jeanne,
and my children, Jennifer and Kathryn.*

Jim Peterson

To Carla.

Peter Galvin

We thank the following people, all of whom reviewed this edition of the book, David Black, Randy Bentson, Joseph Boykin, Wayne Hathaway, Christopher Haynes, Steven Stepanek, Louis Stevens, G. Scott Graham, Thomas LeBlanc, Robert Fowler, and Thomas Casavant.

A.S.

J.P.

P.G.

CONTENTS

PART 1 ■ Overview

CHAPTER □ Introduction

1

- 1.1 What Is an Operating System? 3
- 1.2 Early Systems 6
- 1.3 Simple Monitor 8
- 1.4 Off-Line Operation 11
- 1.5 Buffering and Spooling 14
- 1.6 Multiprogramming 17
- 1.7 Time Sharing 18
- 1.8 Distributed Systems 21
- 1.9 Real-Time Systems 23
- 1.10 Single-User Systems 24
- 1.11 Summary 24
- Exercises 25
- Bibliographic Notes 26

CHAPTER □ Computer System Structures

2

- 2.1 Interrupt-Based Systems 29
- 2.2 I/O Structure 34

2.3	Dual-Mode Operation	36
2.4	Hardware Protection	38
2.5	General System Architecture	42
2.6	Different Classes of Computers	43
2.7	Summary	48
	Exercises	49
	Bibliographic Notes	50

CHAPTER ☐ Operating System Structures

3

3.1	System Components	51
3.2	Operating-System Services	57
3.3	System Calls	59
3.4	System Programs	67
3.5	System Structure	70
3.6	Virtual Machines	75
3.7	System Design and Implementation	79
3.8	System Generation	81
3.9	Summary	83
	Exercises	84
	Bibliographic Notes	84

PART 2 ■ Process Management

CHAPTER ☐ Processes

4

4.1	Process Concept	89
4.2	Concurrent Processes	93
4.3	Scheduling Concepts	97
4.4	CPU Scheduling	103
4.5	Scheduling Algorithms	106
4.6	Multiple Processor Scheduling	121
4.7	Algorithm Evaluation	122
4.8	Summary	128

Exercises	129
Bibliographic Notes	132

CHAPTER Process Coordination

5

5.1	Background	135
5.2	The Critical-Section Problem	139
5.3	Synchronization Hardware	147
5.4	Semaphores	149
5.5	Classical Problems of Synchronization	155
5.6	Language Constructs	160
5.7	Interprocess Communication	175
5.8	Summary	186
	Exercises	186
	Bibliographic Notes	192

CHAPTER Deadlocks

6

6.1	System Model	195
6.2	Deadlock Characterization	197
6.3	Deadlock Prevention	201
6.4	Deadlock Avoidance	204
6.5	Deadlock Detection	211
6.6	Recovery from Deadlock	216
6.7	Combined Approach to Deadlock Handling	218
6.8	Summary	219
	Exercises	220
	Bibliographic Notes	224

PART 3 Storage Management

CHAPTER Memory Management

7

7.1	Background	229
7.2	Swapping	235

7.3	Single-Partition Allocation	238
7.4	Multiple-Partition Allocation	242
7.5	Multiple Base Registers	249
7.6	Paging	250
7.7	Segmentation	260
7.8	Paged Segmentation	267
7.9	Summary	269
	Exercises	270
	Bibliographic Notes	274

CHAPTER Virtual Memory

8

8.1	Motivation	275
8.2	Demand Paging	277
8.3	Performance of Demand Paging	284
8.4	Page Replacement	286
8.5	Page-Replacement Algorithms	290
8.6	Allocation of Frames	301
8.7	Thrashing	304
8.8	Other Considerations	310
8.9	Demand Segmentation	318
8.10	Summary	319
	Exercises	320
	Bibliographic Notes	327

CHAPTER Secondary Storage Management

9

9.1	Background	329
9.2	Disk Structure	330
9.3	Free-Space Management	334
9.4	Allocation Methods	336
9.5	Disk Scheduling	345
9.6	Selecting a Disk-Scheduling Algorithm	350
9.7	Sector Queueing	352
9.8	Performance and Reliability Improvements	353

9.9	Storage Hierarchy	354
9.10	Summary	356
	Exercises	357
	Bibliographic Notes	360

PART 4 ■ Files and Protection

CHAPTER □ File Systems

10

10.1	File-System Organization	365
10.2	File Operations	370
10.3	Access Methods	372
10.4	Consistency Semantics	375
10.5	Directory-Structure Organization	376
10.6	File Protection	386
10.7	Implementation Issues	390
10.8	Summary	392
	Exercises	393
	Bibliographic Notes	394

CHAPTER □ Protection

11

11.1	Goals of Protection	397
11.2	Access Matrix	400
11.3	Dynamic Protection Structures	405
11.4	Revocation	410
11.5	Existing Systems	411
11.6	Language-Based Protection	417
11.7	Protection Problems	422
11.8	Security	425
11.9	Encryption	427
11.10	Summary	430
	Exercises	430
	Bibliographic Notes	432

PART 5 ■ Distributed Systems

CHAPTER □ Distributed System Structures

12

- 12.1 Motivation 437
- 12.2 Topology 439
- 12.3 Communication 444
- 12.4 Network Types 449
- 12.5 Types of Operating Systems 452
- 12.6 Design Issues 456
- 12.7 Summary 458
 - Exercises 459
 - Bibliographic Notes 460

CHAPTER □ Distributed Coordination

13

- 13.1 Event Ordering 463
- 13.2 Mutual Exclusion 466
- 13.3 Deadlock Prevention 470
- 13.4 Deadlock Detection 471
- 13.5 Robustness 480
- 13.6 Reaching Agreement 482
- 13.7 Election Algorithms 485
- 13.8 Summary 488
 - Exercises 488
 - Bibliographic Notes 489

CHAPTER □ Distributed File Systems

14

- 14.1 Background 491
- 14.2 Naming and Transparency 493
- 14.3 Remote Services 497
- 14.4 Caching 499
- 14.5 Stateful versus Stateless Service 505
- 14.6 File Replication 507
- 14.7 Example Systems 508

14.8	Summary	539
	Exercises	540
	Bibliographic Notes	540

PART 6 ■ Case Studies

CHAPTER □ The UNIX Operating System 15

15.1	History	545
15.2	Design Principles	551
15.3	Programmer Interface	553
15.4	User Interface	562
15.5	Process Management	566
15.6	Memory Management	571
15.7	File System	574
15.8	I/O System	583
15.9	Interprocess Communication	587
15.10	Summary	594
	Exercises	594
	Bibliographic Notes	595

CHAPTER □ The Mach Operating System 16

16.1	History	597
16.2	Design Principles	599
16.3	System Components	601
16.4	Process Management	604
16.5	Interprocess Communication	611
16.6	Memory Management	617
16.7	Programmer Interface	623
16.8	Implementation Details	624
16.9	Summary	627
	Exercises	628
	Bibliographic Notes	628

CHAPTER ☐ Historical Perspective

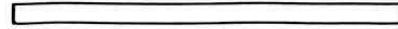
17

- 17.1 Atlas 631
- 17.2 XDS-940 633
- 17.3 THE 633
- 17.4 RC 4000 634
- 17.5 CTSS 636
- 17.6 MULTICS 636
- 17.7 OS/360 637
- 17.8 Other Systems 639

Bibliography 641

Index 679

Introduction



An *operating system* is a program that acts as an intermediary between a 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. The primary goal of an operating system is thus to make the computer system *convenient* to use. A secondary goal is to use the computer hardware in an *efficient* manner.

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 to current multiprogrammed and time-shared 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. Understanding the reasons behind the development of operating systems gives an appreciation for what tasks an operating system does and how it does them.

1.1 What Is an Operating System?

An operating system is an important part of almost every computer system. A computer system can be roughly divided into four components: the *hardware*, the *operating system*, the *applications programs*, and the *users* (Figure 1.1).

The hardware (the central processing unit (CPU), memory, and input/output (I/O) devices) provides the basic computing resources. The

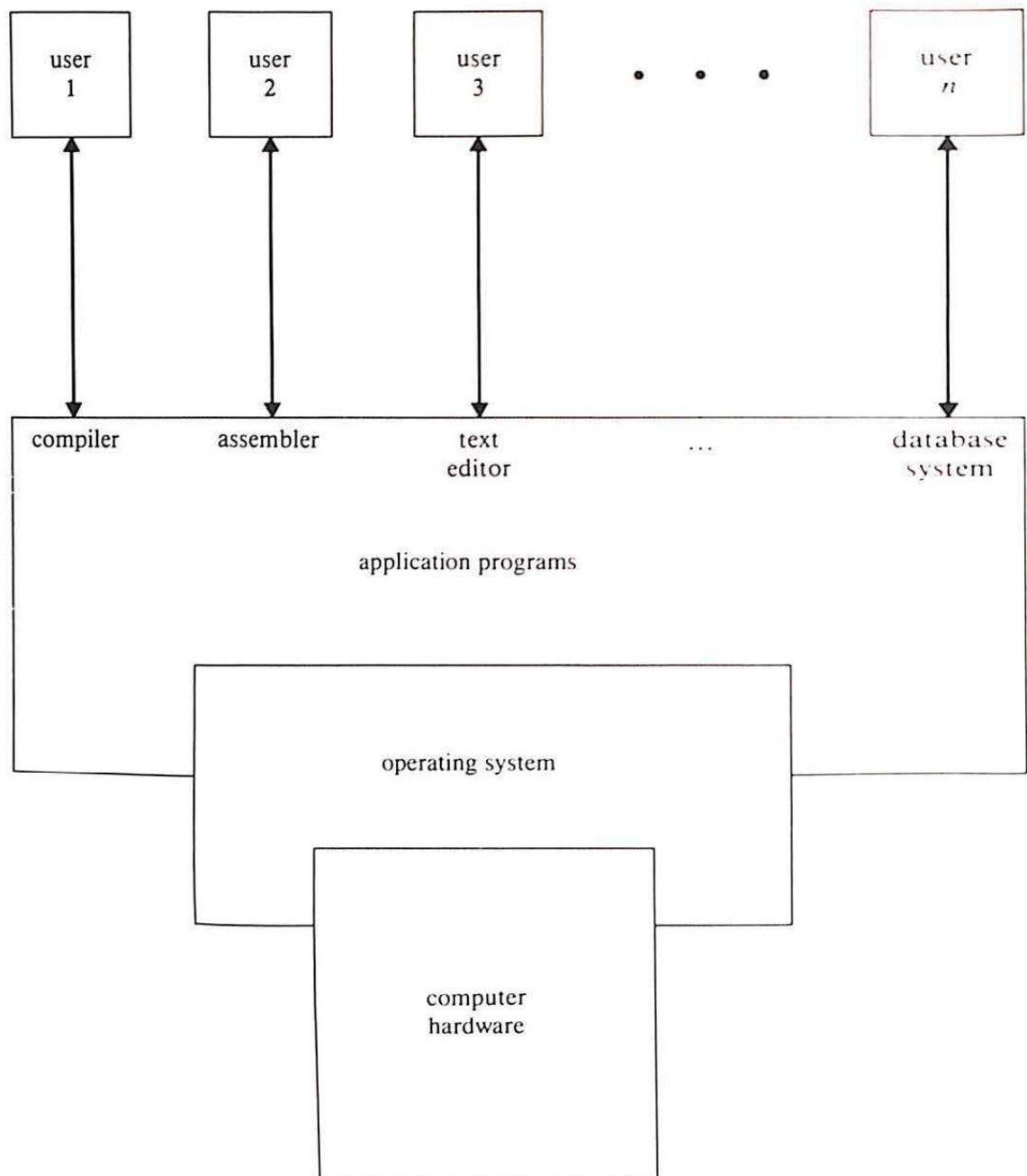


Figure 1.1 Abstract view of the components of a computer system.

applications programs (compilers, database systems, video games, and business programs) define the ways in which these resources are used to solve the computing problems of the users. There may be many different users (people, machines, other computers) trying to solve different problems. Accordingly, there may be many different

applications programs. The operating system controls and coordinates the use of the hardware among the various application programs for the various users.

An operating system is similar to a *government*. 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. *Like a government, the operating system performs no useful function by itself.* It simply provides an *environment* within which other programs can do useful work.

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 and allocates them to specific programs and users as necessary for the latter's tasks. Since there may be many, possibly conflicting, requests for resources, the operating system must decide which requests are allocated resources to operate the computer system efficiently and fairly.

A slightly different view of an operating system focuses on the need to control the various I/O devices and user programs. An operating system is a *control program*. A control program controls 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, there is 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 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 very easy to use, applications programs are developed. These various programs require certain common operations, such as 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.

There is also no universally accepted definition of what is part of the operating system and what is not. A simple viewpoint is that everything a vendor ships when you order "the operating system" should be considered. The memory requirements and features included, however, vary greatly between systems. Some take up less than 1 megabyte of space and lack even a full-screen editor, whereas others require hundreds of megabytes of space and include spelling checkers and entire window systems. 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 application

programs. The latter is more common and is the one we generally follow.

It is perhaps easier to define operating systems by what they *do*, rather than what they *are*. The primary goal of an operating system is *convenience for the user*. Operating systems exist because they are supposed to make it easier to compute with one than without one. This is particularly clear when you look at operating systems for small personal computers.

A secondary goal is *efficient* operation of the computer system. This goal is particularly important for large, shared multiuser systems. These systems are typically very expensive, and so it is desirable to make them as efficient as possible. These two goals, convenience and efficiency, are sometimes contradictory. In the past, efficiency considerations were often more important than convenience. Thus, much of operating-system theory concentrates on optimal use of computing resources.

To see what operating systems are and what operating systems do, let us consider how they have developed over the last 30 years. By tracing that evolution, we can identify the common elements of operating systems and see how and why these systems have developed as they have.

Operating systems and computer architecture have had a great deal of influence on each other. To facilitate the use of the hardware, operating systems were developed. As operating systems were designed and used, it became obvious that changes in the design of the hardware could simplify them. In this short historical review, notice how the introduction of new hardware features is the natural solution to many operating-system problems.

1.2 Early Systems

Initially, there was only computer hardware. Early computers were (physically) very large machines run from a console. The programmer would write a program and then operate the program directly from the operator's console. First, the program would be manually loaded into memory, from the front panel switches (one instruction at a time), from paper tape, or from punched cards. Then the appropriate buttons would be pushed to set the starting address and to start the execution of the program. As the program ran, the programmer/operator could monitor its execution by the display lights on the console. If errors were discovered, the programmer could halt the program, examine the contents of memory and registers, and debug the program directly from the console. Output was printed, or was punched onto paper tape or cards for later printing.

An important aspect of this environment was its *hands-on* interactive nature. The programmer was also the operator of the computer system. Most systems used a sign-up or *reservation scheme* for allocating machine time. If you wanted to use the computer, you went to the sign-up sheet, looked for the next convenient free time on the machine, and signed up for it.

There were, however, certain problems with this approach. Suppose you had signed up for 1 hour of computer time to run a program that you were developing. You might run into a particularly nasty bug and be unable to finish in 1 hour. If someone had reserved the following block of time, you would have to stop, collect what you could, and return at a later time to continue. On the other hand, if your program ran smoothly, you might finish in 35 minutes. Since you had thought you might need the machine longer, you had signed up for 1 hour, and so the machine would now sit idle for 25 minutes.

As time went on, additional software and hardware were developed. Card readers, line printers, and magnetic tape became commonplace. Assemblers, loaders, and linkers were designed to ease the programming task. Libraries of common functions were created. Common functions could then be copied into a new program without having to be written again.

The routines that performed I/O were especially important. Each new I/O device had its own characteristics, requiring careful programming. A special subroutine was written for each I/O device. Such a subroutine is called a *device driver*. A device driver knows how the buffers, flags, registers, control bits, and status bits for a particular device should be used. Each different type of device has its own driver. A simple task, such as reading a character from a paper-tape reader, might involve complex sequences of device-specific operations. Rather than the necessary code being written every time, the device driver was simply used from the library.

Later, compilers for FORTRAN, COBOL, and other languages appeared, making the programming task much easier, but the operation of the computer more complex. To prepare a FORTRAN program for execution, for example, the programmer would first need to load the FORTRAN compiler into the computer. The compiler was normally kept on magnetic tape, so the proper tape would need to be mounted on a tape drive. The program would be read through the card reader and written onto another tape. The FORTRAN compiler produced assembly-language output, which then needed to be assembled. This required mounting another tape with the assembler. The output of the assembler would need to be linked to supporting library routines. Finally, the binary object form of the program would be ready to execute. It could be loaded into memory and debugged from the console, as before.

File Systems

For most users, the file system is the most visible aspect of an operating system. Files store data and programs. The operating system implements the abstract concept of a file by managing mass storage devices, such as tapes and disks. In this chapter, we consider the various ways to map files onto devices. Also, files are normally organized into directories to ease their use, so we look at a variety of directory structures. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed. This control is known as *file protection*.

10.1 File-System Organization

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms; magnetic disk and tape are the most common storage media. Each storage device has its own characteristics and physical organization.

So that the computer system will be convenient to use, 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*. Files are mapped, by the operating system, onto physical devices.

The file system consists of two distinct parts: the collection of the actual *files*, each containing related information, and the *directory structure*, which provides information about all the files in the system.

10.1.1 File Concept

What is a *file*? 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, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records whose meaning is defined by the file's creator and user. The concept of a file is thus very general.

A file is named, and is referred to by its name. It has certain other properties, such as its type, the time of its creation, the name (or account number) of its creator, its length, and so on.

The information in a file is defined by its creator. Many different types of information may be stored in a file: source programs, object programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined *structure* according to its type. A *text* file is a sequence of characters organized into lines (and possibly pages); a *source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements; an *object* file is a sequence of bytes organized into loader record blocks.

One major consideration is whether the operating system should know and support file types. If an operating system knows the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to print the binary object form of a program. This attempt normally produces garbage, but can be prevented *if* the operating system has been told that the file is a binary object program.

Another example comes from the TOPS-20 operating system. If the user tries to execute an object program whose source file has been modified (edited) since the object file was produced, the source file will be recompiled automatically. This function ensures that the user always runs an up-to-date object file. Otherwise, the user could waste a significant amount of time executing the old object file. Notice that, for this function to be possible, the operating system must be able to discriminate the source file from the object file, to check the time that each file was last modified or created, and to determine the language of the source program (in order to use the correct compiler).

There are disadvantages to having the operating system know the type of a file. One problem is the resulting size of the operating system.

If the operating system defines 14 different file types, it must then contain the code to support these file types correctly. In addition, every file must be definable as one of the file types supported by the operating system. Severe problems may result from new applications that require information structured in ways not supported by the operating system.

For example, assume that a system supports two types of files: text files (composed of ASCII characters separated by a carriage return and line feed) and executable binary files. Now if we (as users) want to define an encrypted file to protect our files from being read by unauthorized people, we may find neither file type to be appropriate. The encrypted file is not ASCII text lines, but rather is (apparently) random bits. Although it may appear to be a binary file, however, it is not executable. As a result, we may have to circumvent or misuse the operating system's file-types mechanism, or to modify or abandon our encryption scheme.

The other extreme is to impose (and support) no file type in the operating system. This approach has been adopted in UNIX, among others. UNIX considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system. This scheme provides maximum flexibility, but minimal support. Each application program must include its own code to interpret an input file into the appropriate structure.

Files are usually kept on disks. Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O is in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. *Packing* a number of logical records into physical blocks is a common solution to this problem.

For example, the UNIX operating system defines all files to be simply a stream of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks (say, 512 bytes per block) as necessary.

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system.

In either case, the file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem.

Notice that disk space being always allocated in blocks has the result that, in general, some portion of the last block of each file may be

wasted. If each block is 512 bytes, then a file of 1949 bytes would be allocated four blocks (2048 bytes); the last 99 bytes would be wasted. The wasted bytes allocated to keep everything in units of blocks (instead of bytes) is *internal fragmentation*. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

10.1.2 Directory Structure

The files in a computer system are represented by entries in a *device directory* or *volume table of contents*. The device directory records information — such as name, location, size, and type — for all files on that device.

A device directory may be sufficient for a single-user system with limited storage space. As the amount of storage and the number of users increase, however, it becomes increasingly difficult for the users to organize and keep track of all the files. The solution to this problem is the imposition of a *directory structure* on the file system. A directory structure provides a mechanism for organizing the many files in the file system. It may span device boundaries and include several different disk units, or even span disks on different computers. In this way, the user needs to be concerned with only the logical directory and file structure, and can completely ignore the problems of physically allocating space for files.

In fact, many systems actually have two separate directory structures: the device directory and the file directories. The device directory is stored on each physical device and describes all files on that device. The device-directory entry mainly describes the physical properties of each file: where the file is, how long it is, how it is allocated, and so on. The file directories are a logical organization of the files on all devices. The file-directory entry concentrates on logical properties of each file: name, file type, owning user, accounting information, protection access codes, and so on. A file-directory entry may simply point to the device-directory entry to provide information on physical properties, or may duplicate this information. Our main interest now is with the file-directory structure; device directories should be well understood.

The particular information kept for each file in the directory varies from operating system to operating system. The following information may be kept in a directory entry; not all systems keep all this information:

- **File name.** The symbolic file name is the only information kept in human-readable form.

- **File type.** This information is needed for those systems that support different types.
- **Location.** This is a pointer to the device and location on that device of the file.
- **Size.** The current size of the file (in bytes, words or blocks), and possibly the maximum allowed size are included.
- **Current position.** This is a pointer to the current read or write position in the file.
- **Protection.** Access-control information controls who can do reading, writing, executing, and so on.
- **Usage count.** This value indicates the number of processes that are currently using (have opened) this file.
- **Time, date, and process identification.** This information may be kept for (1) creation, (2) last modification, and (3) last use. These data can be useful for protection and for usage monitoring.

It may take from 16 to over 1000 bytes to record this information for each file. In a system with a large number of files, the size of the directory itself may be hundreds of thousands of bytes. Thus, the device directory may need to be stored on the device and brought into memory piecemeal, as needed.

The directory system can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, then it becomes apparent that the directory itself can be organized in many ways. We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. We discuss the appropriate data structures that can be used in the implementation of the directory structure. In Section 10.5, we discuss logical organization of the directory structure.

A linear list of directory entries requires a linear search to find a particular entry. This is simple to program but is time-consuming in execution. To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory. To delete a file, we search the directory for the named file, then release the space allocated to it. To reuse the directory entry, we can do one of several things. We can mark the entry as unused (a special name, such as an all-blank name, or a used/unused bit in each entry), or we can attach it to a list of free directory entries. A third alternative is to copy the last entry in the directory in the freed location and to decrease the length of the directory. A linked list can also be used to decrease the time to delete a file.

The real disadvantage of a linear list of directory entries is the linear search to find a file. A sorted list allows a binary search and decreases the average search time. However, the search algorithm is more complex to program. In addition, the list must be kept sorted. This requirement may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory. (Notice, however, that if we want to be able to produce a list of all files in a directory sorted by file name, we do not have to sort separately before listing.) A linked binary tree might help here.

Another data structure that has been used for a file directory is a *hash table*. A hash-table data structure can greatly improve the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for *collisions* — situations where two file names hash to the same location. The major difficulties with a hash table are the generally fixed size of the hash table and the dependence of the hash function on the size of the hash table.

For example, assume we establish a hash table of 64 entries. The hash function converts file names into integers from 0 to 63, probably by a final operation that uses the remainder of a division by 64. If we later try to create a sixty-fifth file, we must enlarge the directory hash table — say, to 100 entries. As a result, we need a new hash function, which must map file names to the range 0 to 99, and we must reorganize the existing directory entries to reflect their new hash-function values.

10.2 File Operations

A file is an *abstract data type*. To define a file properly, we need to consider the operations that can be performed on files. The operating system provides system calls to create, write, read, reset, and delete files. Let us consider what the operating system must do for each of the five basic file operations. It should then be easy to see how similar operations, such as renaming a file, would be implemented.

- **Creating a file.** Two steps are necessary to create a file. First, space in the file system must be found for the file. We shall discuss how to allocate space for the file in Section 10.4. Second, an entry for the new file must be made in the directory. The directory entry records the name of the file and the location in the file system.
- **Writing a file.** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the location of the file. The directory entry will need to store a pointer to the current block of the file (usually the beginning of the

file). Using this pointer, we can compute the address of the next block and write the information. The write pointer must be updated. In this way, successive writes can be used to write a sequence of blocks to the file.

- **Reading a file.** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated directory entry, and the directory will need a pointer to the next block to be read. Once that block is read, the pointer is updated.

In general, a file is either being read or written, thus, although it would be possible to have two pointers, a read pointer and a write pointer, most systems have only one, a *current file position*. Both the read and write operations use this same pointer, saving space in the directory entry, and reducing the system complexity.

- **Resetting a file.** The directory is searched for the appropriate entry, and the current file position is reset to the beginning of the file. Resetting a file need not involve any actual I/O.
- **Deleting a file.** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space (so it can be reused by other files) and invalidate the directory entry.

You may have noticed that all the operations mentioned involve searching the directory for the entry associated with the named file. The directory entry contains all the important information needed to operate on the file. To avoid this constant searching, many systems will *open* a file when that file first is used actively. The operating system keeps a small table containing information about all open files. When a file operation is requested, an index into this table is used, so no searching is required. When the file is no longer actively used, it is *closed* and is removed from the table of open files.

Some systems implicitly open a file when the first reference is made to it. The file is automatically closed when the job or program that opened the file terminates. Most systems, however, require that a file be opened explicitly by the programmer with a system call (*open*) before that file can be used. The open operation takes a file name and searches the directory, copying the directory entry into the table of open files, assuming the file protections allow such access. Then, the open system call will typically return a pointer to the entry in the table of open files. This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching.

The five operations we described certainly comprise the minimal set of required file operations. More commonly, we shall also want to *edit* the file and to modify its contents. A common modification is *appending* new information to the end of an existing file. We may want to create a *copy* of a file, or copy the file to an I/O device, such as a printer or a display. Since files are named objects, we may want to *rename* an existing file. We certainly want to execute a file if it is object format. Also of use are facilities to lock sections of an open file for multiprocess access, to share sections, and even to map sections into memory on virtual-memory systems. This last function allows a part of the virtual address space to be logically associated with a section of a file. Reads and writes to that memory region are then treated as reads and writes to the file, greatly simplifying file use.

10.3 Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. There are several ways that the information in the file can be accessed. Some systems provide only one access method for files. On other systems, such as those of IBM, many different access methods are supported, and choosing the right one for a particular application is a major design problem.

10.3.1 Sequential Access

Information in the file is processed in order, one record after the other. This is by far the most common mode of access of files. For example, computer editors usually access files in this fashion.

The bulk of the operations on a file are reads and writes. A read operation reads the *next* portion of the file and automatically advances the file pointer. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning, and, on some systems, a program may be able to skip forward or backward n records, for some integer n (perhaps only for $n = 1$). This scheme is known as *sequential access* to a file (Figure 10.1). Sequential access is based on a tape model of a file.

10.3.2 Direct Access

An alternative access method is *direct access*, which is based on a disk model of a file. For direct access, the file is viewed as a numbered

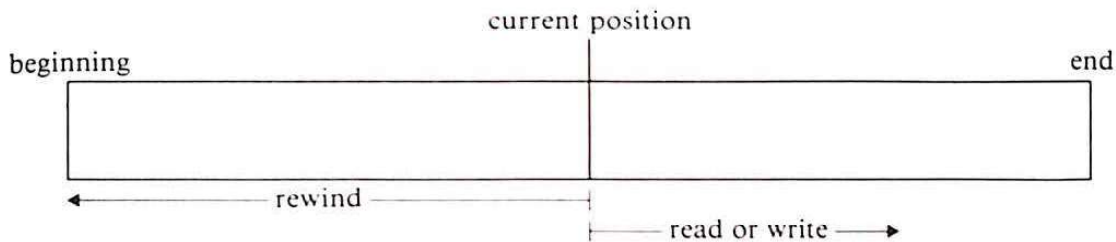


Figure 10.1 Sequential-access file.

sequence of blocks or records. A direct-access file allows arbitrary blocks to be read or written. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct-access files are of great use for immediate access to large amounts of information. Large databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer, and then read that block directly to provide the desired information.

For example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set, such as people, we might compute a hash function on the people's names, or search a small in-core index to determine a block to read and search.

The file operations must be modified to include the block number as a parameter. Thus, we have *read n*, where *n* is the block number, rather than *read next*, and *write n*, rather than *write next*. An alternative approach is to retain *read next* and *write next*, as with sequential access, and to add an operation, *position file to n*, where *n* is the block number. Then, to effect a *read n*, we would *position to n* and then *read next*.

The block number provided by the user to the operating system is normally a *relative block number*. A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block, and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed (the allocation problem as discussed in Section 10.5), and helps to prevent the user from accessing portions of the file system that may not be part of his file. Some systems start their relative block numbers at 0; others start at 1.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created; such a file can be accessed only in a manner consistent with its declaration. Notice, however, that it is easy to simulate sequential access on a direct-access file. If we simply keep a variable *cp*, which defines our current position, then we can simulate sequential file operations, as shown in Figure 10.2. On the other hand, it is extremely inefficient and clumsy to simulate a direct-access file on a sequential-access file.

10.3.3 Other Access Methods

Other access methods can be built on top of a direct-access method. These additional methods generally involve the construction of an *index* for the file. The index, like an index in the back of a book, contains pointers to the various blocks. To find an entry in the file, we first search the index and then use the pointer to access the file directly and to find the desired entry.

For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each entry consists of a 10-digit UPC code and a six-digit price, for a 16-byte entry. If our disk has 1024 bytes per block, we can store 64 entries per block. A file of 120,000 entries would occupy about 2000 blocks (2 million bytes). By keeping the file sorted by UPC code, we can define an index consisting of the first UPC code in each block. This index would have 2000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory. To find the price of a particular item, we can (binary) search the index. From this search, we would know exactly which block contains the desired entry and access that block. This structure allows us to search a large file with very little I/O.

Sequential access	Implementation for direct access
<i>reset</i>	<i>cp</i> := 0;
<i>read next</i>	<i>read cp</i> ;
	<i>cp</i> := <i>cp</i> + 1;
<i>write next</i>	<i>write cp</i> ;
	<i>cp</i> := <i>cp</i> + 1;

Figure 10.2 Simulation of sequential access on a direct-access file.

With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files, which would point to the actual data items.

For example, IBM's indexed sequential access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct access reads.

10.4 Consistency Semantics

Consistency semantics is an important criterion for evaluation of any file system that supports sharing of files. It is a characterization of the system that specifies the semantics of multiple users accessing a shared file simultaneously. In particular, these semantics should specify when modifications of data by one user are observable by other users.

For the following discussion, we assume that a series of file accesses (that is, Reads and Writes) attempted by a user to the same file, is always enclosed between the Open and Close operations. We call the series of accesses between an Open and Close operation a *file session*. To illustrate the concept, we sketch several prominent examples of consistency semantics.

10.4.1 UNIX Semantics

The UNIX file system (Chapter 15) uses the following consistency semantics.

- Writes to an open file by a user are visible immediately to other users that have this file open at the same time.
- There is a mode of sharing where users share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.

These semantics lend themselves to an implementation where a file is associated with a single physical image that is accessed as an exclusive

resource. Contention for this single image results in users being delayed.

10.4.2 Session Semantics

The Andrew file system (Chapter 14) uses the following consistency semantics.

- Writes to an open file by a user are not visible immediately to other users that have the same file open simultaneously.
- Once a file is closed, the changes made to it are visible only in sessions starting later. Already-open instances of the file do not reflect these changes.

According to these semantics, a file may be temporarily associated with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both Read and Write accesses concurrently on their image of the file, without being delayed. Notice that almost no constraints are enforced on scheduling accesses.

10.4.3 Immutable-Shared-Files Semantics

A different, quite unique approach is that of *immutable shared files*. Once a file is declared as shared by its creator, it cannot be modified any more. An immutable file has two important properties: Its name may not be reused and its contents may not be altered. Thus, the name of an immutable file signifies the fixed contents of the file, not the file as a container for variable information. The implementation of these semantics in a distributed system (Chapter 14) is very simple, since the sharing is very disciplined (read-only).

10.5 Directory-Structure Organization

Many different file directory structures have been proposed and are in use. The directory is essentially a *symbol table*. The operating system takes the symbolic file name and finds the named file. In this section, we examine several alternative schemes for defining the logical structure of the directory system. When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

- **Search.** We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and

similar names may indicate a relationship between files, we may want to be able to find all files that match a particular pattern.

- **Create file.** New files need to be created and added to the directory.
- **Delete file.** When a file is no longer needed, we want to remove it from the directory.
- **List directory.** We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Backup.** For reliability, it is generally a good idea to save the contents and structure of the file system at regular intervals. This saving often consists of copying all files to magnetic tape. This technique provides a backup copy in case of system failure or if the file is simply no longer in use. In this case, the file can be copied to tape, and the disk space of that file released for reuse by another file.

10.5.1 Single-Level Directory

The simplest directory structure is the single-level directory. The device directory is an example of a single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 10.3).

A single-level directory has significant limitations, however, when the number of files increases or when there is more than one user. Since all files are in the same directory, they must have unique names. If we have two users who call their data file *test*, then the unique-name rule is

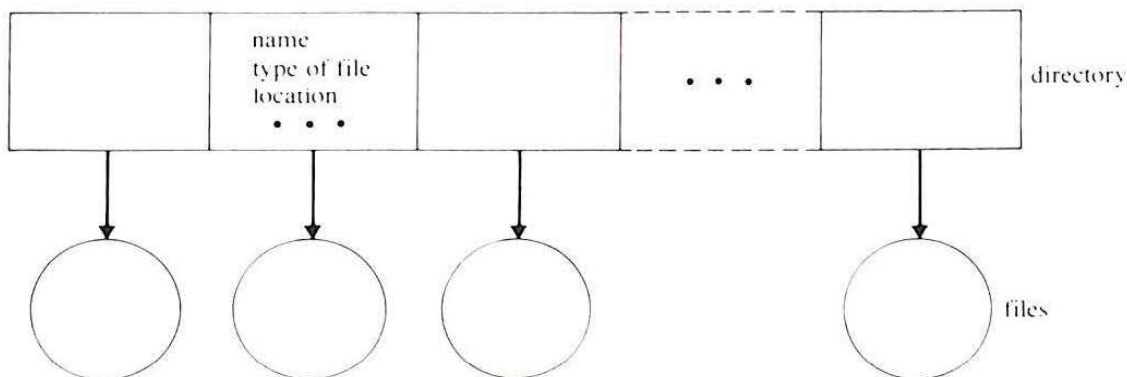


Figure 10.3 Single-level directory.

violated. (For example, in one programming class, 23 students called the program for their second assignment *prog2*; another 11 called it *assign2*.) Although file names are generally selected to reflect the content of the file, they are often quite limited in length. (MS-DOS allows only 11-character file names; UNIX allows 255 characters.)

Even with a single user, as the number of files increases, it becomes difficult to remember the names of all the files in order to create only files with unique names. (It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system.)

10.5.2 Two-Level Directory

The major disadvantage to a single-level directory is the confusion of file names between different users. The standard solution is to create a *separate* directory for each user. Especially on a large system, this user directory is a *logical*, rather than physical, organization, since all the files are still physically on the same device.

In the two-level directory structure, each user has her own user file directory (UFD). Each UFD has a similar structure (linear, binary, or hashed), but lists only the files of a single user. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The master file directory is indexed by user name or account number, and each entry points to the user directory for that user (Figure 10.4).

When a user refers to a particular file, only his own user file directory is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.

To create a file for a user, the operating system searches only that user's directory to ascertain whether another file of that name already

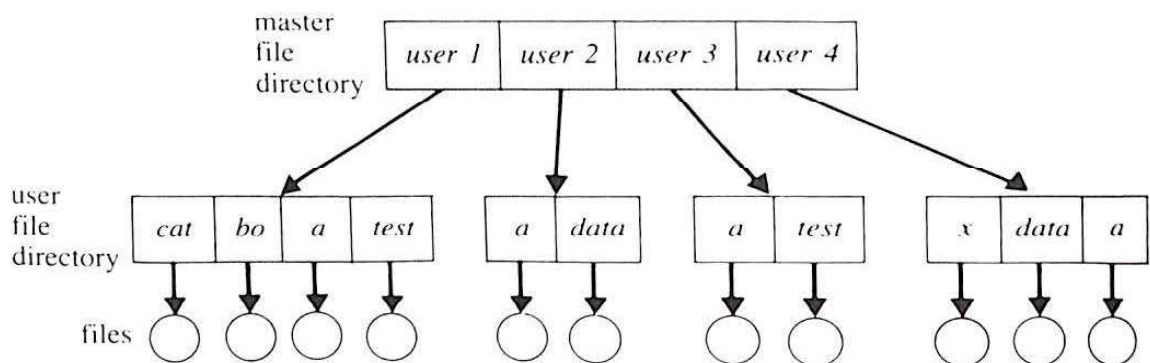


Figure 10.4 Two-level directory structure.

exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file with the same name.

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new user file directory and adds an entry for it to the master file directory. The execution of this program might, of course, be restricted. The allocation of disk space for user directories can be handled using the techniques discussed in Section 10.5 for files themselves.

There are still problems with the two-level directory structure. This structure effectively isolates one user from another. This is an advantage when the users are completely independent, but a disadvantage when the users *want* to cooperate on some task and to access files of other users. Some systems simply do not allow local user files to be accessed by other users.

If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file uniquely in a two-level directory, we must give both the user name and the file name. A two-level directory can be thought of as a tree of height 2. The root of the tree is the master file directory. Its direct descendants are the user file directories. The descendants of the user file directories are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the master file directory) to a leaf (the specified file). Thus, a user name and a file name define a *path name*. Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.

For example, if user *A* wishes to access her own test file named *test*, she can simply refer to *test*. To access the test file of user *B* (with directory-entry name *userb*), however, she might have to refer to */userb/test*. Every system has its own syntax for naming files in directories other than the user's own.

A special case of this situation occurs in regard to the system files. Those programs provided as a part of the system (loaders, assemblers, compilers, utility routines, libraries, and so on) are generally defined as files. When the appropriate commands are given to the operating system, these files are read by the loader and are executed. Many command interpreters act by simply treating the command as the name of a file to load and execute. As the directory system is presently defined, this file name would be searched for in the current user file directory. One solution would be to copy the system files into each user file directory. However, copying all the system files would be enormously wasteful of space. (If the system files require 5 megabytes,

then supporting 12 users would require sixty ($5 \times 12 = 60$) megabytes just for copies of the system files.)

The standard solution is to complicate the search procedure slightly. A special user directory is defined to contain the system files (for example, user 0). Whenever a file name is given to be loaded, the operating system first searches the local user file directory. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files. The sequence of directories searched when a file is named is called the *search path*.

10.5.3 Tree-Structured Directories

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (Figure 10.5). This generalization allows users to create their own subdirectories and to organize their files accordingly. The MS-DOS file system, for instance, is structured as a tree. The tree has a root directory. Every file in the system has a unique path name. A

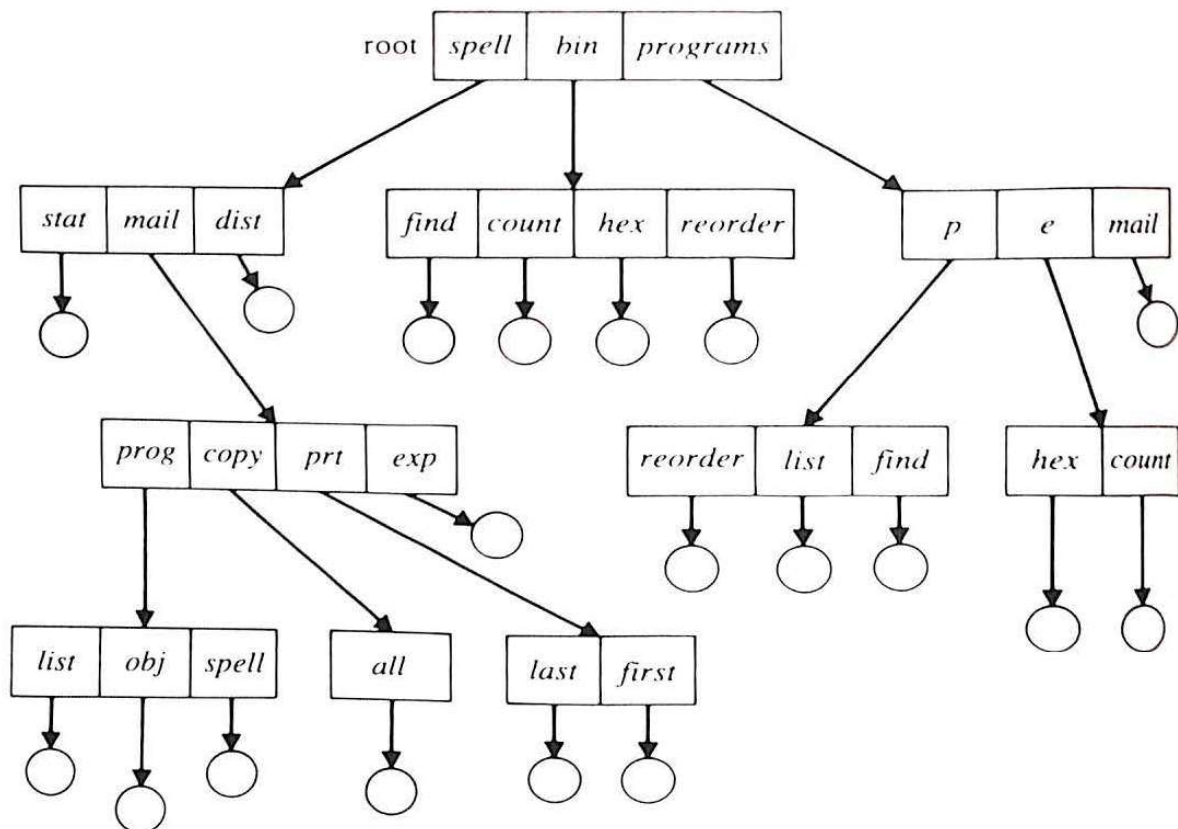


Figure 10.5 Tree-structured directory structure.

path name is the path from the root, through all the subdirectories, to a specified file.

A directory (or subdirectory) contains a set of files and/or subdirectories. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or a subdirectory (1). Special system calls are used to create and delete directories.

In normal use, each user has a *current directory*. The current directory should contain most of the files that are of current interest to the user. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user must either specify a path name or change the current directory. A system call will take a directory name as a parameter and use it to redefine the current directory. Thus, the user can change his current directory whenever he desires. When the user job starts, or the user logs in, the operating system searches the accounting file to find an entry for this user (for accounting purposes). Also stored in the accounting file is a pointer to (or the name of) the user's initial directory. A local variable for this user is defined, to specify the user's current directory.

Path names can be of two types: *absolute* path names or *relative* path names. An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path. A relative path name defines a path from the current directory. For example, in the tree structured file system of Figure 10.5, if the current directory is *root/spell/mail*, then the relative path name *prt/first* refers to the same file as does the absolute path name *root/spell/mail/prt/first*.

Allowing the user to define his own subdirectories permits him to impose a structure on his files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book) or different forms of information (for example, the directory *programs* may contain source programs; the directory *bin* may store all the binaries).

An interesting policy decision in a tree-structured directory structure is how to handle the deletion of a directory. If a directory is empty, its entry in its containing directory can simply be deleted. However, suppose the directory to be deleted is not empty, but contains several files, or possibly subdirectories. One of two approaches can be taken. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, someone must first delete all the files in that directory. If there are any subdirectories, this procedure must be applied recursively to them, so that they can be deleted also. This approach may result in a substantial amount of work.

An alternative approach is just to assume that, when a request is made to delete a directory, all of that directory's files and subdirectories

are also to be deleted. Note that either approach is fairly easy to implement; the choice is one of policy. There are systems that have been implemented each way.

Files of other users can be accessed easily. For example, user *B* can access files of user *A* by specifying their path names. User *B* can specify either an absolute or a relative path name. Alternatively, user *B* could change her current directory to be user *A*'s directory, and access the files by their file names. Some systems also allow users to define their own search paths. In this case, user *B* could define her search path to be (1) her local directory, (2) the system file directory, and (3) user *A*'s directory, in that order. As long as the name of a file of user *A* did not conflict with the name of a local file or system file, it could be referred to simply by its name.

10.5.4 Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. The common subdirectory should be *shared*. A shared directory or file will exist in the file system in two (or more) places at once. Notice that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, there is only *one* actual file, so any changes made by one person would be immediately visible to the other. This is particularly important for shared subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

A tree structure prohibits the sharing of files or directories. An *acyclic graph* allows directories to have shared subdirectories and files (Figure 10.6). The *same* file or subdirectory may be in two different directories. An acyclic graph (that is, a graph with no cycles) is a natural generalization of the tree-structured directory scheme.

In a situation where several people are working as a team, all the files to be shared may be put together into one directory. The user file directories of all the team members would each contain this directory of shared files as a subdirectory. Even when there is a single user, his file organization may require that some files be put into several different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project.

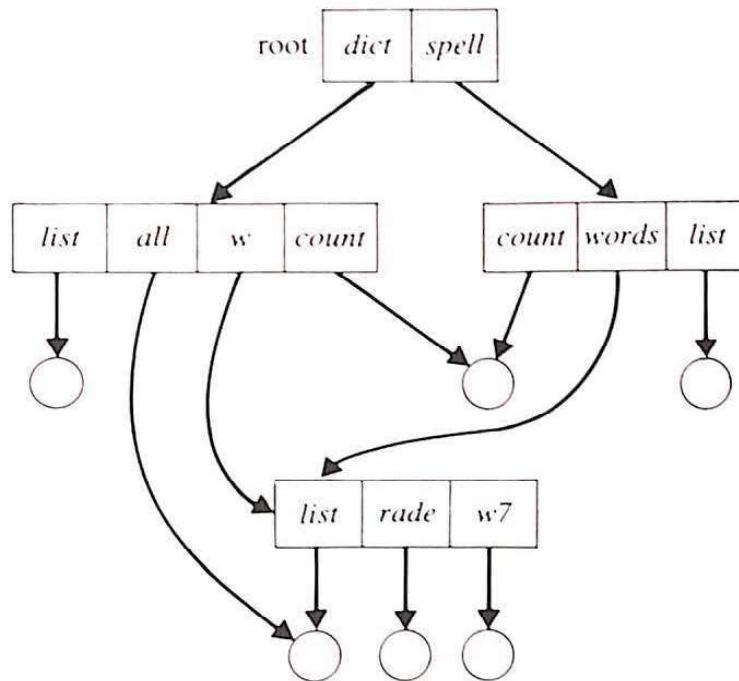


Figure 10.6 Acyclic-graph directory structure.

Shared files and subdirectories can be implemented in several ways. A common way, exemplified by BSD UNIX, is to create a new directory entry called a *link*. A link is effectively a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or relative path name (a *symbolic link*). When a reference to a file is made, we search the directory. The directory entry is marked as a link and the name of the real file (or directory) is given. We *resolve* the link by using the path name to locate the real file. Links are easily identified by their format in the directory entry, and are effectively named indirect pointers. The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

The other approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. A link is clearly different from the original directory entry; thus, the two are not equal. Duplicate directory entries, however, make the original and the copy indistinguishable. A major problem with duplicate directory entries is maintaining consistency if the file is modified.

An acyclic-graph directory structure is more flexible than a simple tree structure, but is also more complex. Several problems must be considered carefully. Notice that a file may now have multiple absolute

path names. Consequently, distinct file names may refer to the same file. This is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system (to find a file, to accumulate statistics on all files, or to copy all files to backup storage), this problem becomes significant, since we do not want to traverse shared structures more than once.

Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link need not affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them also, but unless a list of the associated links is kept with each file, this search can be quite expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist, and can fail to resolve the link name; the access is treated just like any other illegal file name. (In this case, the system designer should consider carefully what to do when a file is deleted and another file of the same name is created, before a symbolic link to the original file is used.)

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

The trouble with this approach is the variable and potentially large size of the file-reference list. However, we really do not need to keep the entire list — we need to keep only a count of the *number* of references. A new link or directory entry increments the reference count; deleting a link or entry decrements the count. When the count is 0, the file can be deleted; there are no remaining references to it. UNIX uses this approach, keeping a reference count in the file information block (i-node). By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure. A description of the UNIX file system is provided in Chapter 15.

10.5.5 General Graph Directory

One serious problem with using an acyclic graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves its tree-structured nature. However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure (Figure 10.7).

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse it and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file, without finding that file, we want to avoid searching that subdirectory again; the second search would be a waste of time.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating.

A similar problem exists when we are trying to determine when a file can be deleted. As with acyclic-graph directory structures, a 0 in the

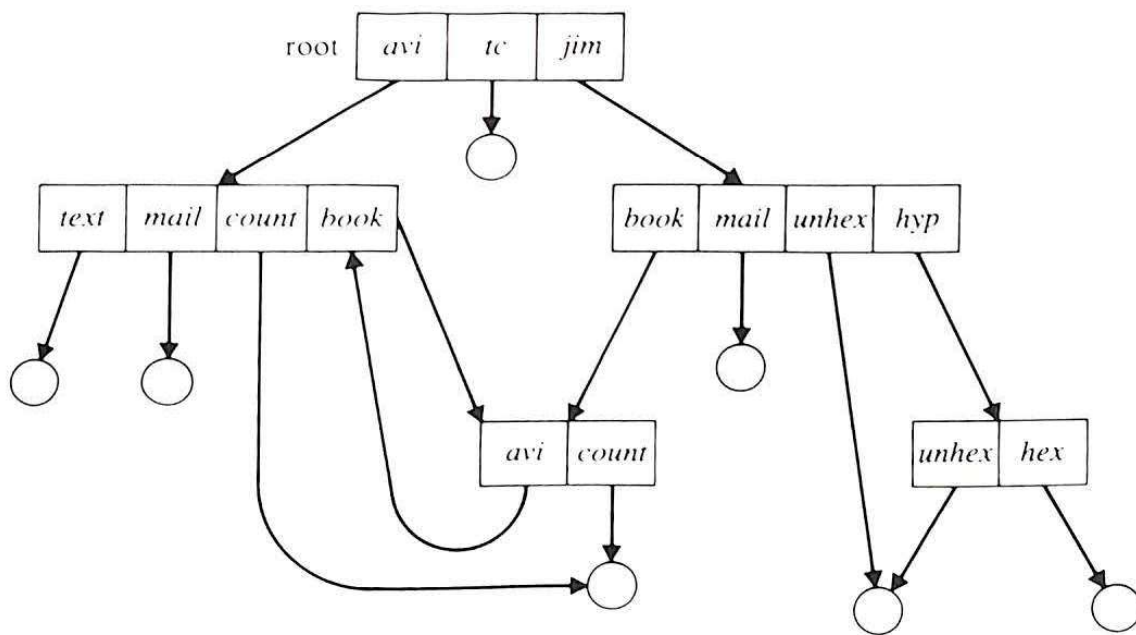


Figure 10.7 General graph directory.

reference count means that there are no more references to the file or directory, and the file can be deleted. However, it is also possible, when cycles exist, that the reference count may be nonzero, even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing (a cycle) in the directory structure. In this case, it is generally necessary to use *garbage collection* to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. (A similar marking procedure can be used to ensure that a traversal or search will cover everything in the file system once and only once.) Garbage collection for a disk-based file system, however, is extremely time-consuming and is thus seldom attempted.

Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles, as new links are added to the structure. How do we know when a new link will complete a cycle? There are algorithms to detect cycles in graphs; however, they are computationally expensive, especially when the graph is on disk storage. Generally, tree directory structures are more common than are acyclic-graph structures.

10.6 File Protection

When information is kept in a computer system, a major concern is its protection from both physical damage (*reliability*) and improper access (*protection*).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature, and vandalism. Files may be deleted accidentally. Bugs in the file system software can also cause file contents to be lost.

Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multiuser system, however, other mechanisms are needed.

The need for protecting files is a direct result of the ability to access files. On systems that do not permit access to the files of other users, protection is not needed. Thus, one extreme would be to provide

complete protection by prohibiting access. The other extreme is to provide free access with no protection. Both of these approaches are too extreme for general use. What is needed is *controlled access*.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read.** Read from the file.
- **Write.** Write or rewrite the file.
- **Execute.** Load the file into memory and execute it.
- **Append.** Write new information at the end of the file.
- **Delete.** Delete the file and free its space for possible reuse.

Other operations, such as renaming, copying, or editing the file, may also be controlled. For many systems, however, these higher-level functions (such as copying) may be implemented by a system program that makes lower-level system calls. Protection is provided at only the lower level. For instance, copying a file may be implemented simply by a sequence of read requests. In this case, a user with read access can also cause the file to be copied, printed, and so on.

The directory operations that must be protected are somewhat different. We want to control the creation and deletion of files in a directory. In addition, we probably want to control whether a user can determine the existence of a file in a directory. Sometimes, knowledge of the existence and name of a file may be significant in itself. Thus, listing the contents of a directory must be a protected operation.

Many different protection mechanisms have been proposed. Chapter 11 fully addresses protection issues; we shall discuss only a few pertinent ones. As always, each has its advantages and disadvantages and must be selected as appropriate for its intended application. A small computer system that is used by only a few members of a research group may not need the same types of protection as will a large corporate computer that is used for research, finance, and personnel operations.

Protection can be associated either with the file itself or with the path used to specify the file. The more common scheme provides protection on the path. Thus, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic or general graphs), a given user may have different access rights to a file depending on the path name used.

10.6.1 Naming

The protection schemes of several systems depend on the inability of a user to access a file that he cannot name. If the user cannot name a file, then he cannot operate on it. This scheme assumes that there is no mechanism for obtaining the names of other user's files and that the names cannot be guessed easily. In fact, since file names are generally picked to be mnemonic, they can often be guessed easily.

10.6.2 Passwords

Another approach is to associate a password with each file. Just as access to the computer system itself is often controlled by a password, access to each file can be controlled by a password. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file to only those users who know the password.

There are, however, several disadvantages to this scheme:

- If we associate a separate password with each file, the number of passwords that need to be remembered may become large, making the scheme impractical. If only one password is used for all the files, then, once it is discovered, all files are accessible. Some systems (for example, TOPS-20) allow a user to associate a password with a subdirectory rather than with an individual file, to deal with this problem.
- Commonly, only one password is associated with each file. Thus, protection is on an all-or-nothing basis. To provide protection on a more detailed level, we must use multiple passwords.

10.6.3 Access Lists

Another approach is to make access dependent on the identity of the user. Various users may need different types of access to a file or directory. An *access list* can be associated with each file and directory, specifying the user name and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs and the user job is denied access to the file.

10.6.4 Access Groups

The main problem with access lists can be their length. If we want to allow everyone to read a file, we must list all users with read access.

This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry that previously was of fixed size needs now to be of variable size, resulting in space management being more complicated.

These problems can be resolved by use of a condensed version of the access list.

To condense the length of the access list, many systems recognize three classifications of users in connection with each file:

- **Owner.** The user who created the file is the owner.
- **Group.** A set of users who are sharing the file and need similar access is a group, or workgroup.
- **Universe.** All other users in the system comprise the universe.

As an example, consider a person, Sara, who is designing a new computer game. She has hired three graduate students (Jim, Dawn, and Jill) to work on the project. The game program is kept in a file named *game*. The protection associated with this file is as follows:

- Sara should be able to invoke all operations on the file.
- Jim, Dawn, and Jill should be able only to read, write, and execute the file; they should not be allowed to delete the file.
- All others users should be able to execute the file. (Sara is interested in letting as many people as possible play the game in order to obtain appropriate feedback.)

To achieve such a protection, we must create a new group with members Jim, Dawn, and Jill. The name of the group must be then associated with the file *game*, and the access-right must be set in accordance with the policy we have outlined.

Note that, for this scheme to work properly, group membership must be tightly controlled. This control can be accomplished in a number of different ways. For example, in the UNIX system, groups can be created and modified by only the manager of the facility (or by any superuser). Thus, this control is achieved through human interaction. In the VMS system, groups can be created and controlled just as files are, with appropriate control.

With this classification, only three fields are needed to define protection. Each field is often a collection of bits, each of which either allows or prevents the access associated with it. For example, the UNIX system defines three fields of 3 bits each: *rw**x*, where *r* controls read access, *w* controls write access, and *x* controls execution. A separate field is kept for the file owner, for the owner's group and for all other users. In this scheme, 9 bits per file are needed to record protection information.

10.7 Implementation Issues

A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves the definition of a file and its attributes, operations allowed on a file, and the directory structure. Next, algorithms and data structures must be created to map the logical file system onto the physical devices that exist for use with the file system.

The file system itself is generally composed of many different levels, as shown in Figure 10.8. This is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

The lowest level, *I/O control*, consists of device drivers and interrupt handlers to transfer information between memory and the disk system. The *basic file system* uses this information to read and write particular blocks to and from the disk. Each disk block is identified by its numeric disk address (for example, drive 1, cylinder 73, surface 2, sector 10).

The *file organization module* knows about both files and disk blocks. By knowing the type of file allocation used and the location of the file, the file organization module can generate the addresses of the blocks for the basic file system to read. Finally, the *logical file system* uses the directory structure to provide the file-organization module with the values the latter needs from a symbolic file name.

To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, we read the appropriate directory into memory, update it with the new entry, and write it back to the disk. The directories can be treated exactly as files are — files with a type field indicating that they are directories. Thus, the logical file system can call the file-organization module to map the directory I/O into disk-block numbers, which are passed on to the basic file system and I/O control system.

Once the directory has been updated, the logical file system can use it to perform I/O. When a file is opened, the directory structure is

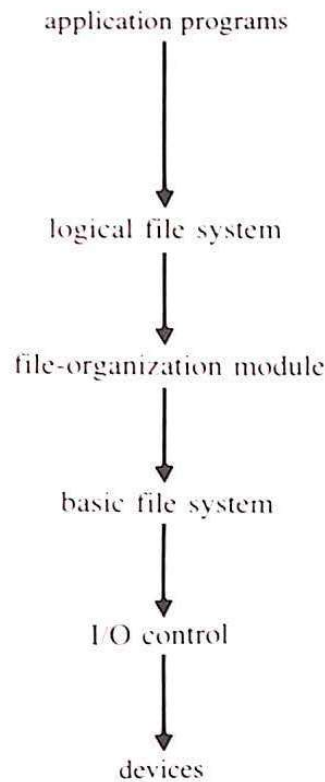


Figure 10.8 Layered file system.

searched for the desired file entry. It would be possible to search the directory structure for every I/O operation, but that would be inefficient. To speed the search, the operating system generally keeps a table of opened files in memory. The first reference to a file (normally an open) causes the directory structure to be searched and the directory entry for this file to be copied into the table of opened files. The index into this table is returned to the user program and all further references are made through the index (a file descriptor) rather than with a symbolic name. All changes to the directory entry are made to the copy in the table of opened files. When the file is closed, the updated entry is copied back to the disk-based directory structure.

While this approach is much faster than are constant references to the disk copy of the directory entry, it can cause problems. If the operating system fails, the table of opened files is generally lost, and with it any changes in the directories of opened files. This can leave the file system in an inconsistent state, where the actual state of some files is not as described in the directory structure. Frequently, a special program is run at reboot time to check for and correct disk inconsistencies.

Another aspect of file use that must be defined carefully is access control (protection). Every access to a file or directory must be checked for correctness. Systems have been designed that check file protection only when the file is opened. Separate system calls are used for **open for read** and **open for write**. Although the operating system can certainly check protection at this point, it must continue to check that a file that is **open for read** is never written to. Thus, *every* access must be checked for validity.

10.8 Summary

A file is an abstract data type defined and implemented by the operating system. A file is a sequence of logical records. A logical record may be a byte, a line (fixed or variable length), or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.

The major task for the operating system is to map the logical file concept onto physical storage devices such as magnetic tape or disk. Since the physical record size of the device may not be the same as the logical record size, it may be necessary to block logical records into physical records. Again, this task may be supported by the operating system or left for the application program.

Tape-based file systems are quite constrained; most file systems are disk-based. Tapes are commonly used for data transport between machines, or for backup or archival storage.

Each device in a file system keeps a volume table of contents or device directory listing the location of the files on the device. In addition, it is useful to create directories to allow files to be organized. A single-level directory in a multiuser system causes naming problems, since each file must have a unique name. A two-level directory solves this problem by creating a separate directory for each user. Each user has her own directory, containing her own files.

The natural generalization of a two-level directory is a tree-structured directory. A tree-structured directory allows a user to create subdirectories to organize his files. Acyclic-graph directory structures allow subdirectories and files to be shared, but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories, but sometimes requires garbage collection to recover unused disk space.

Since files are the main information-storage mechanism in most computer systems, file protection is needed. Access to files can be controlled separately for each type of access: read, write, execute,

append, list directory, and so on. File protection can be provided by passwords, by access lists, or by special ad hoc techniques.

File systems are often implemented in a layered or modular structure. The lower levels deal with the physical properties of storage devices. Upper levels deal with symbolic file names and logical properties of files. Intermediate levels map the logical file concepts into physical device properties.

Exercises

- 10.1 Consider a file system where a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area? How can these problems be avoided?
- 10.2 Some systems automatically delete all user files when a user logs off or a job terminates, unless the user explicitly requests that they be kept; other systems keep all files unless the user explicitly deletes them. Discuss the relative merits of each approach.
- 10.3 Could you simulate a multilevel directory structure with a single-level directory structure in which arbitrarily long names can be used? If your answer is yes, explain how you can do so, and contrast this scheme with the multilevel directory scheme. If your answer is no, explain what prevents your simulation's success. How would your answer change if file names were limited to seven characters?
- 10.4 Explain the purpose of the *open* and *close* operations.
- 10.5 Some systems automatically open a file when it is referenced for the first time, and close the file when the job terminates. Discuss the advantages and disadvantages of this scheme as compared to the more traditional one, where the user has to open and close the file explicitly.
- 10.6 Give an example of an application in which data in a file should be accessed in the following order:
 - a. Sequentially
 - b. Randomly
- 10.7 Some systems provide file sharing by maintaining a single copy of a file; other systems maintain several copies, one for each of

the users sharing the file. Discuss the relative merits of each approach.

- 10.8 In some systems, a subdirectory can be read and written by an authorized user, just as ordinary files can be.
 - a. Describe what protection problems that could arise.
 - b. Suggest a scheme for dealing with each of the protection problems you named in part a.
- 10.9 Consider a system that supports 5000 users. Suppose that you want to allow 4990 of these users to be able to access one file.
 - a. How would you specify this protection scheme in UNIX?
 - b. Could you suggest another protection scheme that can be used more effectively for this purpose than the scheme provided by UNIX?
- 10.10 Researchers have suggested that, instead of having an access list associated with each file (which users can access the file, and how), we should have a *user control list* list associated with each user (which files a user can access, and how). Discuss the relative merits of these two schemes.

Bibliographic Notes

General discussions concerning file systems are offered by Mullender and Tanenbaum [1984], Golden and Pechura [1986], and Grosshans [1986].

The issue of file size is discussed by Powell [1977] in connection to the Demos file system, and by Satyanarayanan [1981] in connection with the TOP-10 system.

A multilevel directory structure was first implemented on the MULTICS system [Organick 1972]. Other systems using a multilevel directory structure include Atlas [Wilkes 1975], TENEX [Bobrow et al. 1972], and UNIX [Ritchie and Thompson 1974]. A flat directory structure for the Pilot, a personal-computer operating system, is described by Redell et al. [1980]. Watson [1970] and Organick [1972] discussed the implementation of directory structures.

File-system integrity was discussed by Wilkes [1975] for the Cambridge multiple-access system and by Organick [1972] for the MULTICS system. The BSD UNIX operating system uses the FSCK program to recover from disk inconsistencies caused by a crash. The process is

described in [CSRG 1986]. MS-DOS uses the CHKDSK program, as documented in [Microsoft 1986]. Anyanwu and Marshall [1986] present a crash resistant UNIX file system. A fast file system for UNIX is presented by McKusick et al. [1984].

The MS-DOS file system is described in [Norton and Wilton 1988]. The Networked File System (NFS) was designed by Sun Microsystems, and allows directory structures to be spread across networked computer systems. Discussions concerning NFS are presented in [Sandberg et al. 1985], [Sandberg 1987], and [Sun Microsystems 1990]. NFS is fully described in Chapter 14. The immutable shared files semantics is described by Schroeder et al. [1985].