

FOURTH EDITION

OPERATING SYSTEM CONCEPTS



Abraham Silberschatz
University of Texas

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

Sponsoring Editor: *Deborah Lafferty*
Senior Editor: *Tom Stone*
Senior Production Supervisor: *Helen Wythe*
Marketing Manager: *Phyllis Cerys*
Technical Art Coordinator: *Susan London-Payne*
Cover and Endpaper Designer: *Howard S. Friedman*
Manufacturing Manager: *Roy Logan*

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 procedures 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.

Library of Congress Cataloging-in-Publication Data

Silberschatz, Abraham.

Operating system concepts / Abraham Silberschatz, Peter B. Galvin.

p. cm.

Includes bibliographical references and index.

ISBN 0-201-50480-4

1. Operating systems (Computers) I. Galvin, Peter B. II. Title.

QA76.76.063S5583 1994

005.4'3--dc20

93-24415

CIP

Reproduced by Addison-Wesley from camera-ready copy supplied by the authors.

Reprinted with corrections June, 1994

Copyright © 1994 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.

ISBN 0-201-50480-4

3 4 5 6 7 8 9 10-MA-9594

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

Avi Silberschatz

To Carla and Gwendolyn.

Peter Galvin

CHAPTER 2

COMPUTER-SYSTEM STRUCTURES



We need to have a general knowledge of the structure of a computer system before we can explore the details of system operation. In this chapter, several disparate parts of this structure are presented to round out our background knowledge. This chapter is mostly concerned with computer-system architecture, so you can skim or skip it if you already understand the concepts. Because an operating system is intimately tied to the I/O mechanisms of a computer, I/O is discussed first. The following sections discuss the data-storage structure.

The operating system must also ensure the correct operation of the computer system. So that user programs will not interfere with the proper operation of the system, the hardware must provide appropriate mechanisms to ensure correct behavior. Later in this chapter, we describe the basic computer architecture that makes it possible to write a functional operating system.

2.1 ■ Computer-System Operation

A modern, general-purpose computer system consists of a CPU and a number of device controllers that are connected through a common bus that provides access to shared memory (Figure 2.1). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, and video displays). The CPU and the device controllers can execute concurrently, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory.

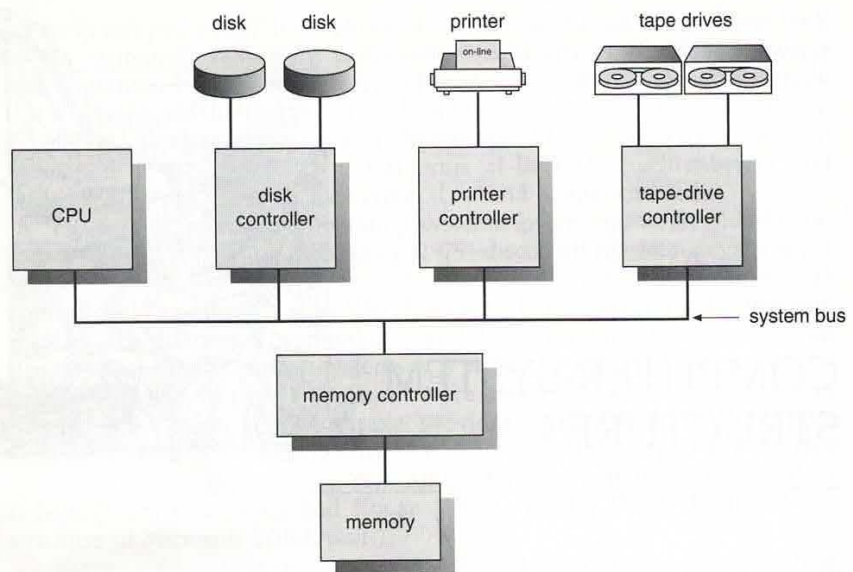


Figure 2.1 A modern computer system.

For a computer to start running — for instance, when it is powered up or rebooted — it needs to have an initial program to run. This initial program, or *bootstrap program*, tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and to start it executing. To accomplish this goal, the bootstrap program must locate the operating system kernel and load it into memory. The operating system then starts executing the first process, such as “init”, and waits for some event to occur. The occurrence of an event is usually signaled by an *interrupt* from either the hardware or software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a *system call* (also called a *monitor call*).

There are many different types of events that may trigger an interrupt, for example, the completion of an I/O operation, division by zero, invalid memory access, and a request for some operating system service. For each such interrupt, a service routine is provided that is responsible for dealing with the interrupt.

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located.

The interrupt service routine executes, and upon completion, the CPU resumes the interrupted computation. A time line of this operation is shown in Figure 2.2.

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for handling this transfer would be to invoke a generic routine to examine the interrupt information, and it, in turn, would call the interrupt-specific handler. However, interrupts must be handled very quickly, and given that there are a predefined number of possible interrupts, a table of pointers to interrupt routines may be used instead. The interrupt routine is then called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first 100 or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or *interrupt vector*, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as MS-DOS and UNIX dispatch interrupts in this manner.

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

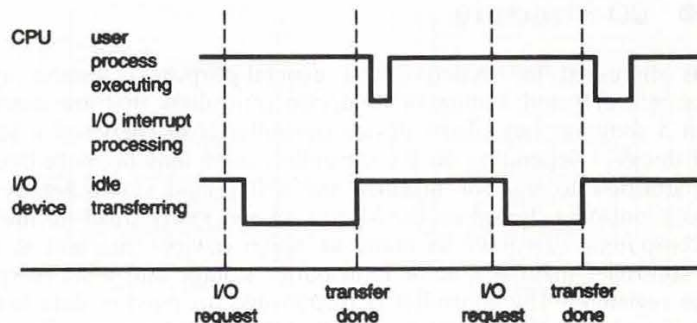


Figure 2.2 Interrupt time line for a single process doing output.

Usually, interrupts are *disabled* while an interrupt is being processed, delaying any incoming interrupts until the operating system is done with the current one, after which interrupts are *enabled*. If they were not thus disabled, the processing of the second interrupt while the first was being serviced would overwrite the first's data, and the first would be a *lost interrupt*. Sophisticated interrupt architectures allow for one interrupt to be processed during another. They often use a priority scheme in which request types are assigned priorities according to their relative importance, and interrupt processing information is stored separately for each priority. A higher-priority interrupt will be taken even if a lower-priority interrupt is active, but interrupts at the same or lower levels are *masked*, or selectively disabled, to prevent lost interrupts or unnecessary ones.

Modern operating systems are *interrupt driven*. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt, or a trap. A *trap* (or an exception) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access), or by a specific request from a user program that an operating-system service be performed.

The interrupt-driven nature of an operating system defines that system's general structure. When an interrupt (or trap) occurs, the hardware transfers control to the operating system. First, the operating system preserves the state of the CPU by storing registers and the program counter. Then, it determines which type of interrupt has occurred. This determination may require *polling*, the querying of all I/O devices to detect which requested service, or it may be a natural result of a vectored interrupt system. For each type of interrupt, separate segments of code in the operating system determine what action should be taken.

2.2 ■ I/O Structure

As was discussed in Section 2.1, a general-purpose computer system consists of a CPU and a number of device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device. Depending on the controller, there may be more than one device attached to it. For instance the *SCSI* (Small Computer Systems Interface) controller, found on the Macintosh and many small- to medium-sized computers, can have as many as seven devices attached to it. A device controller maintains some local buffer storage and a set of special-purpose registers. The controller is responsible for moving data between the peripheral device(s) it controls and its local buffer storage. The size of the local buffer within a device controller varies from one controller to another, depending on the particular device being controlled. For

example, the size of the buffer of a disk controller is the same as or a multiple of the size of the smallest addressable portion of a disk (a *sector*), which is usually 512 bytes.

2.2.1 I/O Interrupts

To start an I/O operation, the CPU loads the appropriate registers within the device controller. The device controller, in turn, examines the contents of these registers to determine what action to take. For example, if it finds a read request, the controller will start the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the CPU that it has finished its operation. It accomplishes this communication by causing an interrupt.

This situation will occur, in general, as the result of a user process requesting I/O. Once the I/O is started, two courses of action are possible. In the simplest case, the I/O is started; then, at I/O completion, control is returned to the user process. This is known as *synchronous I/O*. The other possibility (*asynchronous I/O*) is to return control to the user program without waiting for the I/O to complete. The I/O then can continue while other system operations occur (see Figure 2.3).

Waiting for I/O completion may be accomplished in one of two ways. Some computers have a special **wait** instruction that idles the CPU until the next interrupt. Machines that do not have such an instruction may have a wait loop:

Loop: **jmp** Loop

This very tight loop simply continues until an interrupt occurs, transferring

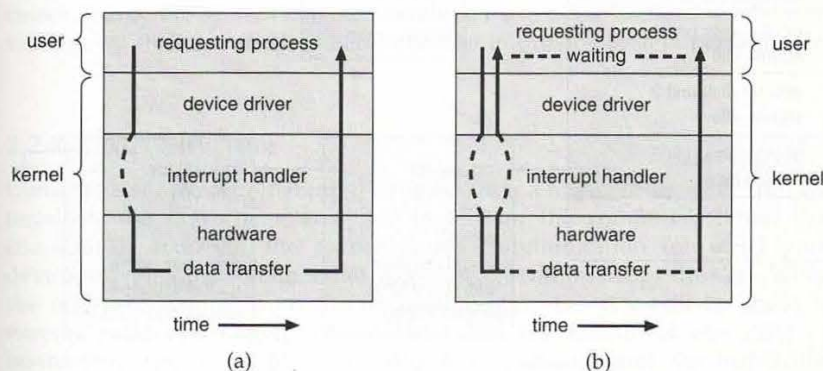


Figure 2.3 Two I/O methods. (a) synchronous. (b) asynchronous.

control to another part of the operating system. Such a loop might also need to poll any I/O devices which do not support the interrupt structure; instead they simply set a flag in one of their registers and expect the operating system to notice it. The *wait* instruction is probably better than a loop, since a wait loop generates a series of instruction fetches, which may cause significant contention for memory access. The contention is caused by the I/O device transferring information and the CPU transferring instructions.

One major advantage of always waiting for I/O completion is that at most one I/O request is outstanding at a time. Thus, whenever an I/O interrupt occurs, the operating system knows exactly which device is interrupting. On the other hand, this approach excludes simultaneous I/O processing.

An alternative is to start the I/O and immediately to return control to the user program. A *system call* (a request to the operating system) is then needed to allow the user to wait for I/O completion, if desired. Hence, we still require the wait code that we needed before. We also need to be able to keep track of many I/O requests at the same time. For this purpose, the operating system uses a table containing an entry for each I/O device: the *device-status table* (Figure 2.4). Each table entry indicates the device's type, its address, and its state (not functioning, idle, or busy). If the device is busy with a request, the type of request and other parameters will be stored in the table entry for that device. Since it is possible for other processes to issue requests to the same device, we may have a list or chain

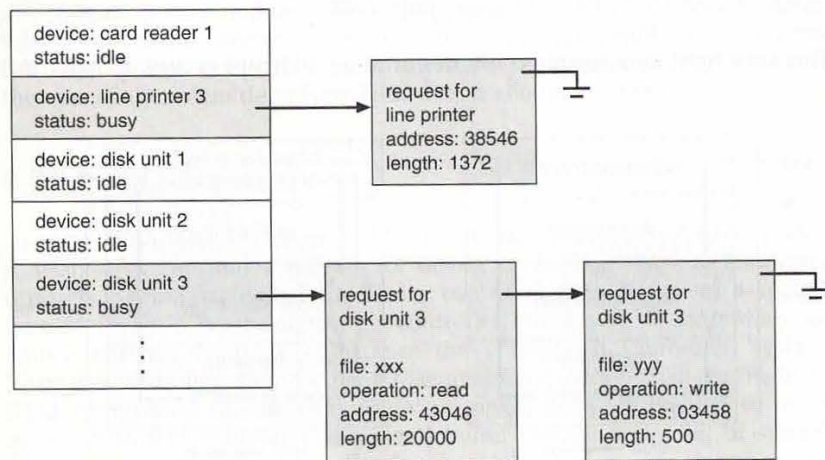


Figure 2.4 Device-status table.

of waiting requests. Thus, in addition to the I/O device table, an operating system may have a request list for each device.

An I/O device interrupts when it needs service. When an interrupt occurs, the operating system first determines which I/O device caused the interrupt. It then indexes into the I/O device table to determine the status of that device, and modifies the table entry to reflect the occurrence of the interrupt. For most devices, an interrupt signals completion of an I/O request. If there is an additional request waiting for this device, the operating system starts processing that request.

Finally, control is returned from the I/O interrupt. If a process was waiting for this request to complete (as recorded in the device-status table), we may now return control to it. Otherwise, we return to whatever we were doing before the I/O interrupt: to the execution of the user program (the program started an I/O operation and that operation has now finished, but the program has not yet waited for the operation to complete) or to the wait loop (the program started two or more I/O operations and is waiting for a particular one to finish, but this interrupt was from one of the others). In a time-sharing system, the operating system could switch to another ready-to-run process.

The schemes used by some input devices may vary from this one. Many interactive systems allow users to type ahead, or to enter data before the data are requested, on their terminal. In this case, interrupts may occur, signaling the arrival of characters from the terminal, while the device-status block indicates that no program has requested input from this device. If typeahead is to be allowed, then a buffer must be provided to store the typeahead characters until some program wants them. In general, we may need a buffer for each input terminal.

The main advantage of asynchronous I/O is the increased system efficiency. While I/O is taking place, the system CPU can be used for processing, or even scheduling other I/O. Because I/O can be quite slow compared to processor speed, the system makes much better use of its facilities. In the next section, we shall see another mechanism for improving system performance.

2.2.2 DMA Structure

Consider a simple terminal input driver. When a line is to be read from the terminal, the first character typed is sent to the computer. When that character is received, the asynchronous communication (or serial port) device to which the terminal line is connected will interrupt the CPU. When the interrupt request from the terminal arrives, the CPU will be about to execute some instruction. (If the CPU is in the middle of executing an instruction, the interrupt is normally held pending until the instruction execution is complete.) The address of this interrupted instruction is saved,

and control is transferred to the interrupt service routine for the appropriate device.

The interrupt service routine saves the contents of any CPU registers it will need to use. It checks for any error conditions that might have resulted from the last input operation. It then takes the character from the device, and stores that character in a buffer. The interrupt routine must also adjust pointer and counter variables, to be sure that the next input character will be stored at the next location in the buffer. The interrupt routine next sets a flag in memory indicating to the other parts of the operating system that new input has been received. The other parts are responsible for processing the data in the buffer, and for transferring the characters to the program requesting input (see Section 2.5). Then, the interrupt service routine restores the contents of any saved registers, and transfers control back to the interrupted instruction.

If characters are being typed to a 9600-baud terminal, the terminal can accept and transfer one character approximately every 1 millisecond, or 1000 microseconds. A well-written interrupt service routine to input characters into a buffer may require 2 microseconds per character, leaving 998 microseconds out of every 1000 for CPU computation (and servicing of other interrupts). Given this disparity, asynchronous I/O is usually assigned a low interrupt priority, allowing other, more important interrupts to be processed first, or even to preempt the current interrupt for another. A high-speed device, however, such as a tape, disk, or communications network, may be able to transmit information at close to memory speeds; the CPU would need 2 microseconds to respond to each interrupt, with interrupts arriving every 4 microseconds (for example). This would not leave much time for process execution.

To solve this problem, direct memory access (DMA) is used for high-speed I/O devices. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data to or from its own buffer storage to memory directly, with no intervention by the CPU. Only one interrupt is generated per block, rather than the one interrupt per byte (or word) generated for low-speed devices.

The basic operation of the CPU is the same. A user program, or the operating system itself, may request data transfer. The operating system finds a buffer (an empty buffer for input, or a full buffer for output) from a queue of buffers for the transfer. (A buffer is typically 128 to 4096 bytes, depending on the device type.) The DMA controller then has its registers set to the appropriate source and destination addresses, and transfer length. This register setting is usually done by a *device driver*, which knows exactly how this information is to be provided to the controller. The DMA controller is then instructed (via control bits in a control register) to start the I/O operation. Meanwhile, the CPU has been free to perform other tasks since it gave the transfer information to the controller. The DMA controller interrupts the CPU when the transfer has been completed.

As an example of the utility of DMA, consider a typical IBM PC. The PC supports interrupt-based I/O, as well as DMA channels. The backup program, which is included with the MS-DOS PC operating system, uses only interrupt-based I/O to copy data between a hard disk and a floppy disk. Several companies have written similar programs that take advantage of DMA data transfer, and the result is a several-fold increase in backup speed.

2.3 ■ Storage Structure

For a computer to do its job of executing programs, the programs must be in main memory. Main memory is the only large storage area that the processor can access directly. It is an array of words or bytes, ranging in size from hundreds of thousands to hundreds of millions. Each word has its own address. Interaction is achieved through a sequence of **load** or **store** instructions to specific memory addresses. The load instruction moves a word from main memory to an internal register within the CPU, while the store instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution.

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

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

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

Thus, most computer systems provide *secondary storage* as an extension of main memory. The main requirement of secondary storage is thus to be able to hold extremely large numbers of data permanently.

The most common secondary-storage device is a magnetic disk, which provides storage of both programs and data. Most programs (compilers, assemblers, sort routines, editors, formatters, and so on) are stored on a disk until loaded into memory. They then use the disk as both the source and destination of the information for their processing. Hence, the proper management of disk storage is of central importance to a computer system, as will be discussed in Chapter 12.

In a larger sense, however, the storage structure that we have described — consisting of registers, main memory, and magnetic disks — is only one of many possible storage systems. There are also cache memory, optical disks, magnetic tapes, and so on. Each storage system provides the basic functions of storing a datum, and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, cost, size, and volatility. In Sections 2.3.1 to 2.3.4, we describe the important storage systems.

2.3.1 Main Memory

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. (Consider that there are machine instructions which take memory addresses as arguments, but none that take disk addresses.) Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

In the case of I/O, as mentioned in Section 2.1, each I/O controller includes registers to hold commands and the data being transferred. Usually, special I/O instructions allow data transfers between these registers and system memory. To allow more convenient access to I/O devices, some systems, including the IBM PC and Apple Macintosh, provide *memory-mapped I/O*. In this case, ranges of memory addresses are set aside, and are physically mapped to the device registers. Reads and writes to these memory addresses cause the data to be transferred to and from the device registers. This method is appropriate for devices with fast response times, such as video controllers. In the IBM PC, each location on the screen is mapped to a memory location. Displaying text on the screen is almost as easy as writing the text into the appropriate memory-mapped locations.

Memory-mapped I/O is also convenient for frequently used devices, such as a serial port. So that a system can communicate with another computer over a modem and telephone line, characters are read and written to a one-byte location in memory. Note that polling is again needed in this circumstance, with the computer looping, constantly checking to see whether a character is available for reading. Interrupts for signaling the availability of new input could be used instead, depending on the hardware configuration.

Given that registers are built into the CPU, they are accessible within one cycle of the CPU clock. The CPU can decode an instruction and perform the given operation on a register's contents all in the same clock tick. The same cannot be said for main memory, which may be located on a system bus and take several cycles to access. In this case, the processor normally needs to *stall* while waiting for the access to complete. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential is called a *cache*, as described in Section 2.4.1.

2.3.2 Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Physically, disks are relatively simple (Figure 2.5). Each disk *platter* has a flat circular shape, like a phonograph record. Its two surfaces are covered with a magnetic material, similar to magnetic tape. Information is recorded on the surfaces.

When the disk is in use, a drive motor spins it at high speed (usually 60 revolutions per second). There is a read-write head positioned just above the surface of the platter. The disk surface is logically divided into *tracks*, which are subdivided into *sectors*. We store information by

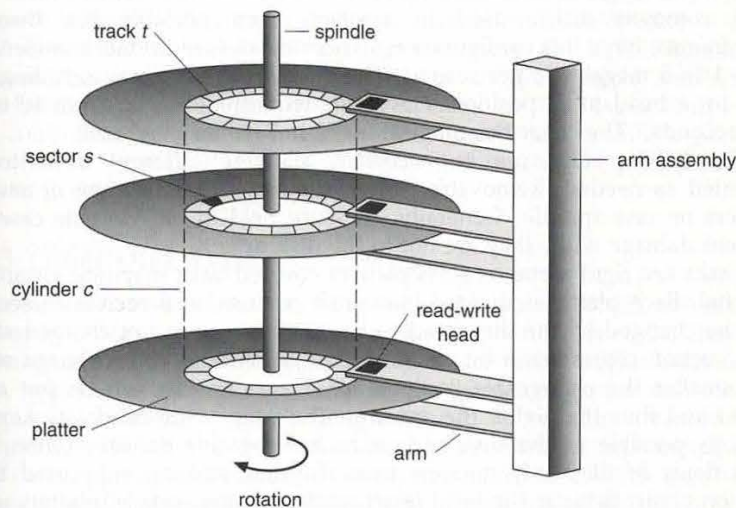


Figure 2.5 Moving-head disk mechanism.

recording it magnetically on the sector under the read-write head. There may be hundreds of concentric tracks on a disk surface, containing thousands of sectors. The platter itself may be between 1.8 inches and 14 inches wide. The larger sizes are common on large systems because of their higher storage capacities and transfer rates. The smaller sizes are found on PCs, since they have lower cost.

A *fixed-head disk* has a separate head for each track. This arrangement allows the computer to switch from track to track quickly, but it requires a large number of heads, making the device extremely expensive. Much more common, is only one head, which moves across the disk to access different tracks. This *moving-head disk*, or simply *hard disk*, requires hardware to move the head, but only a single head is needed, resulting in a much less expensive system. The disk platters, mounted on a spindle and surrounded by heads driven by a motor, are known as *head-disk assemblies* and come in a complete package.

Disks were originally designed for file storage, so the primary design criteria were cost, size, and speed. To provide additional storage capacity, developers took several approaches. They made the primary gain by improving the recording density, allowing more bits to be put on a surface. The density is reflected by the number of tracks per inch, sectors per track, and bits per sector. In addition, with separate heads on each side of the platter, disk capacity can be doubled at minimal cost. We can extend this approach by stacking several disks, each with two recording surfaces, on one spindle. Since all the disks rotate together, only one drive motor is needed, although each surface still needs its own read-write head. The most common disks, used in systems from portable PCs through mainframes, have this configuration. These disks vary in data transfer rate from 1 to 5 megabytes per second. The average access time, including the time for a head to be positioned over the requested data, is from 10 to 40 milliseconds. The capacities range from 10 to 7500 megabytes.

Finally, the disk can be *removable*, allowing different disks to be mounted as needed. Removable disk packs may consist of one or several platters on one spindle. Generally, they are held in hard plastic cases to prevent damage while they are not in the disk drive.

Disks are rigid metal or glass platters covered with magnetic recording material. Each platter is divided into small sections, and each such section may be changed by the disk head to be in a charged or not charged state. Each section represents a bit and is 0 or 1 depending on its charge state. The smaller the changeable sections are, the more bits can be put on a platter and thus the higher the density. The read-write heads are kept as close as possible to the disk surface to increase this density. Often, the head floats or flies only microns from the disk surface, supported by a cushion of air. Because the head floats so close to the surface, platters must be machined carefully to be flat.

Head crashes can be a problem. If the head contacts the disk surface (due to a power failure, for example), the head will scrape the recording medium off the disk, destroying the data that had been there. Usually, the head touching the surface causes the removed medium to become airborne and to come between the other heads and their platters, causing more crashes. Under normal circumstances, a head crash results in the entire disk failing and needing to be replaced.

Floppy disks take a different approach. The disks are coated with a hard surface, so the read-write head can sit directly on the disk surface without destroying the data. Thus, the disk itself is much less expensive to produce and use. The floppy disk must rotate much more slowly than a hard disk, due to the resulting friction. Also, the coating (and the read-write head) will wear with use, and need to be replaced eventually. Because they are more rugged, floppy disks are removable. The disks are not permanently mounted in a head-disk assembly. Instead, the disk is slipped manually into a slot which contains a spindle to rotate it, and a head and motor to access it. This arrangement keeps the cost of floppy disks low because one drive can be used to access hundreds or thousands of disks.

Floppy disks usually have a much lower capacity than do hard disks because they have much lower storage densities, have only one platter, and spin slower. They hold from 100 kilobytes (a kilobyte is 1024 bytes) to a few megabytes per disk. They come in many variations (single-sided, double-sided, single-density, and double-density) and sizes (5 1/4 inch, 3 1/2 inch, and so on). Generally, they are formatted and used in the same way as are hard disks, except that all floppy disks are removable and may therefore be used conveniently to transfer data between computers.

A disk drive has a *disk controller* that determines the logical interaction between the device and the computer. The controller takes instructions from the CPU and orders the disk drive to carry out the instruction. Some disk controllers have a built-in cache, which holds data recently read from or written to the disk. If data are currently in the cache, the need for a disk transfer is obviated.

2.3.3 Other Disk Types

There are many variations on magnetic disks among which we do not distinguish, from an operating-system point of view. Generally, they are treated as normal hard disks except at the device-driver level. There is the *drum*, which has a head over each track. Previously, these fast devices were used for *backing store* (see Chapters 8 and 9), but they are now used rarely due to the improved speed and price-performance ratio of standard hard disks. An up-and-coming device is the *optical disk*, which uses lasers to melt and fill holes on a plastic medium. These drives have been slower but less costly and more rugged than magnetic hard disks in the past, but

their performance is increasing. Because they are more rugged, they, like floppy disks, have the added advantage of removability.

2.3.4 Magnetic Tapes

Magnetic tape was used as an early secondary-storage media. Although it is relatively permanent, and can hold large numbers of data, magnetic tape is quite slow in comparison to the access time of main memory. Even more important, magnetic tape is limited to sequential access. Thus, it is unsuitable for providing the random access needed for most secondary-storage requirements. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool, and is wound or rewound past a read-write head. Moving to the correct spot on a tape can take minutes rather than milliseconds; once positioned, however, tape drives can write data at densities and speeds approaching those of disk drives. Capacities vary depending on the length and width of the tape, and on the density at which the head can read and write. A tape drive is usually named by its width. Thus, there are 8-millimeter, 1/4-inch, and 1/2-inch (also known as 9-track) tape drives. The 8-millimeter tape drives have the highest density, due to the technology they use; they currently store 5 gigabytes of data (a gigabyte is one billion bytes) on a 350-foot tape.

2.4 ■ Storage Hierarchy

The wide variety of storage systems in a computer system can be organized in a hierarchy (Figure 2.6) according to their speed and their cost. The higher levels are expensive, but are fast. As we move down the hierarchy, the cost per bit decreases, whereas the access time increases. This tradeoff is reasonable; if a given storage system were both faster and less expensive than another — other properties being the same — then there would be no reason to use the slower, more expensive memory. In fact, many early storage devices, including paper tape and core memories, are relegated to museums now that magnetic tape and semiconductor memory have become faster and cheaper.

In addition to the speed and cost of the various storage systems, there is also the issue of storage volatility. Volatile storage loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to nonvolatile storage for safekeeping. In the hierarchy shown in Figure 2.6, the storage system above disks is volatile, whereas the storage system below main memory is nonvolatile. The design of a complete memory system attempts to balance all these factors: It uses only as much expensive memory as

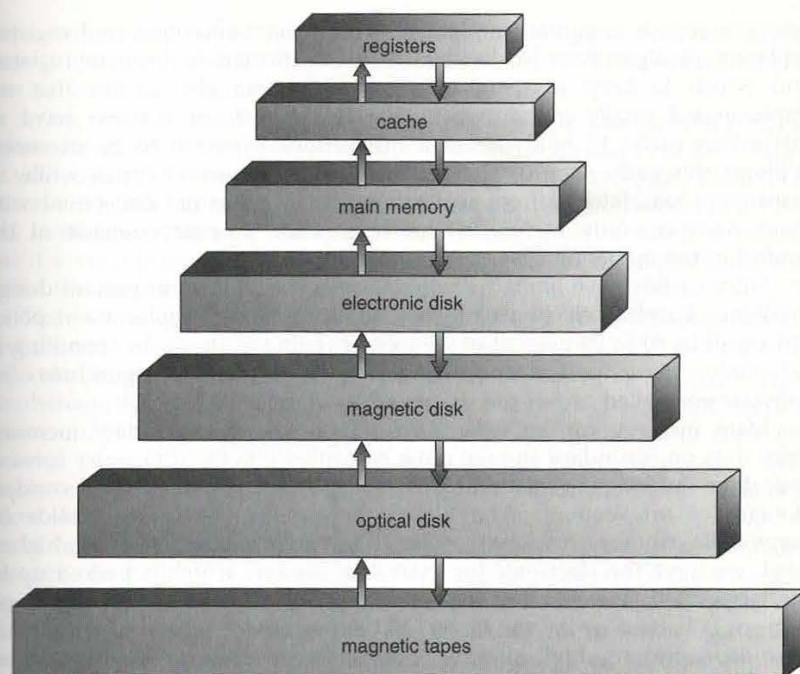


Figure 2.6 Storage-device hierarchy.

necessary, while providing as much inexpensive, nonvolatile, memory as possible. Caches can be installed to ameliorate performance differences where there is a large access-time or transfer-rate disparity between two components.

2.4.1 Caching

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

Extending this view, internal programmable registers, such as index registers and accumulators, are a high-speed cache for main memory. The

programmer (or compiler) implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory. There are also caches that are implemented totally in hardware. For instance, most systems have an instruction cache to hold the next instructions expected to be executed. Without this cache, the CPU would have to wait several cycles while an instruction was fetched from main memory. We are not concerned with these hardware-only caches in this text, since they are outside of the control of the operating system.

Since caches have limited size, *cache management* is an important design problem. Careful selection of the cache size and of a replacement policy can result in 80 to 99 percent of all accesses being in the cache, resulting in extremely high performance. Various replacement algorithms for software-controlled caches are discussed in Chapter 9.

Main memory can be viewed as a fast *cache* for secondary memory, since data on secondary storage must be copied into main memory for use, and data must be in main memory before being moved to secondary storage for safekeeping. The file system itself, which must reside on nonvolatile storage, may have several levels of storage. At the highest level, we have the electronic (or RAM) disk storage, which is backed up by the larger, but slower, magnetic-disk storage. The magnetic-disk storage, in turn, is backed up by the larger, but slower, tape storage. Optical disks are also efficient high-capacity but low-cost storage media. When compared to magnetic tape, they have the drawback of higher cost, but offer much greater speed and convenience. Transfers between these two storage levels are generally requested explicitly, but some systems now automatically archive a file that has not been used for a long time (for example, 1 month), and then automatically fetch back the file to disk when it is next referenced.

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

2.4.2 Coherency and Consistency

In a hierarchical storage structure, the same datum may appear in different storage systems. For example, consider an integer *A* located in file *B* that is to be incremented by 1. Suppose that file *B* resides on magnetic disk. The increment operation proceeds by first issuing an I/O operation to copy the disk block on which *A* resides to main memory. This operation is followed by a possible copying of *A* to the cache, and by copying *A* to an internal register. Thus, the copy of *A* appears in several places. Once the

increment takes place in the internal register, the value of A differs in the various storage systems. The value of A becomes the same only after the new value of A is written back to the magnetic disk.

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

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

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

2.5 ■ Hardware Protection

Early computer systems were single-user programmer-operated systems. When the programmers operated the computer from the console, they had complete control over the system. As operating systems developed, however, this control was given to the operating system. Starting with the resident monitor, the operating system began performing many of the functions, especially I/O, for which the programmer had been responsible previously.

In addition, to improve system utilization, the operating system began to *share* system resources among several programs simultaneously. With *spooling*, one program might have been executing while I/O occurred for other processes; the disk simultaneously held data for many processes. Multiprogramming put several programs in memory at the same time.

This sharing created both improved utilization and increased problems. When the system was run without sharing, an error in a program could cause problems for only the one program that was running. With sharing, many processes could be adversely affected by a bug in one program.

For example, consider the earliest resident monitor, providing nothing more than automatic job sequencing (Section 1.3). Suppose a program gets stuck in a loop reading input cards. The program will read through all its data and, unless something stops it, will continue reading the cards of the next job, and the next, and so on. This loop could prevent the correct operation of many jobs.

Even more subtle errors could occur in a multiprogramming system, where one erroneous program might modify the program or data of another program, or even the resident monitor itself. MS-DOS and Macintosh OS both allow this kind of error.

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

Many programming errors are detected by the hardware. These errors are normally handled by the operating system. If a user program fails in some way — such as an attempt either to execute an illegal instruction, or to access memory that is not in the user's address space — then the hardware will trap to the operating system. The trap transfers control through the interrupt vector to the operating system just like an interrupt. Whenever a program error occurs, the operating system must abnormally terminate the program. This situation is handled by the same code as is a user-requested abnormal termination. An appropriate error message is given, and the memory of the program is dumped. In a batch system, the memory dump may be printed, allowing the user to try to find the cause of the error by examining the printed dump. In an interactive system, the memory dump may be written to a file. The user may then examine it on-line, and perhaps correct and restart the program.

2.5.1 Dual-Mode Operation

To ensure proper operation, we must protect the operating system and all other programs and their data from any malfunctioning program. Protection is needed for any shared resource. The approach taken is to provide hardware support to allow us to differentiate among various modes of executions. At the very least, we need two separate *modes* of operation: *user mode* and *monitor mode* (also called *supervisor mode*, *system mode*, or *privileged mode*). A bit, called the *mode bit*, is added to the hardware of the computer to indicate the current mode: monitor (0) or user (1). With the mode bit, we are able to distinguish between an execution that is done on behalf of the operating system, and one that is done on behalf of the user. As we shall see, this architectural enhancement is useful for many other aspects of system operation.

At system boot time, the hardware starts in monitor mode. The operating system is then loaded, and starts user processes in user mode.

Whenever a trap or interrupt occurs, the hardware switches from user mode to monitor mode (that is, changes the state of the mode bit to be 0). Thus, whenever the operating system gains control of the computer, it is in monitor mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users, and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as *privileged instructions*. The hardware allows privileged instructions to be executed only in monitor mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction, but rather treats the instruction as illegal and traps to the operating system.

The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system. For instance, MS-DOS was written for the Intel 8088 architecture, which has no mode bit, and therefore no dual mode. A user program running awry can wipe out the operating system by writing over it with data, and multiple programs are able to write to a device at the same time, with possibly disastrous results. More recent and advanced versions of the Intel CPU, such as the 80486, do provide dual-mode operation. As a result, more recent operating systems, such as Microsoft Windows/NT, and IBM OS/2, take advantage of this feature and provide greater protection for the operating system.

2.5.2 I/O Protection

A user program may disrupt the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by refusing to relinquish the CPU. Various mechanisms are used to ensure that such disruptions cannot take place in the system.

To prevent a user from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system. For I/O protection to be complete, we must be sure that a user program can never gain control of the computer in monitor mode. If it could, I/O protection could be compromised.

Consider the computer executing in user mode. It will switch to monitor mode whenever an interrupt or trap occurs, jumping to the address determined from the interrupt vector. Suppose a user program, as part of its execution, stores a new address in the interrupt vector. This new address could overwrite the previous address with an address in the user program. Then, when a corresponding trap or interrupt occurred, the hardware would switch to monitor mode, and would transfer control through the (modified) interrupt vector to the user program! The user program could gain control of the computer in monitor mode.

2.5.3 Memory Protection

To ensure correct operation, we must protect the interrupt vector from modification by a user program. In addition, we must also protect the interrupt service routines in the operating system from modification. Otherwise, a user program might overwrite instructions in the interrupt service routine with jumps to the user program, thus gaining control from the interrupt service routine that was executing in monitor mode. Even if the user did not gain unauthorized control of the computer, modifying the interrupt service routines would probably disrupt the proper operation of the computer system and of its spooling and buffering.

We see then that we must provide memory protection at least for the interrupt vector and the interrupt service routines of the operating system. In general, however, we want to protect the operating system from access by user programs, and, in addition, to protect user programs from one another. This protection must be provided by the hardware. It can be implemented in several ways, as we shall see in Chapter 8. Here, we outline one such possible implementation.

What is needed to separate each program's memory space is an ability to determine the range of legal addresses that the program may access, and to protect the memory outside that space. We can provide this protection by using two registers, usually a *base* and a *limit*, as illustrated in Figure 2.7. The base register holds the smallest legal physical memory address, and the limit register contains the size of the range. For example, if the base register holds 300040 and limit register is 120900, then the

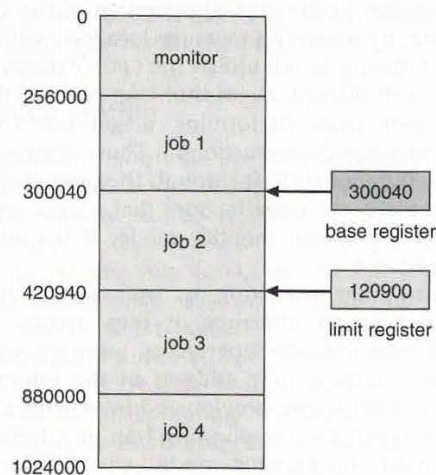


Figure 2.7 A base and a limit register define a logical address space.

program can legally access all addresses from 300040 through 420940 inclusive.

This protection is accomplished by the CPU hardware comparing *every* address generated in user mode with the registers. Any attempt by a program executing in user mode to access monitor memory or other users' memory results in a trap to the monitor, which treats the attempt as a fatal error (Figure 2.8). This scheme prevents the user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The base and limit registers can be loaded by only the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in monitor mode, and since only the operating system executes in monitor mode, only the operating system can load the base and limit registers. This scheme allows the monitor to change the value of the registers, but prevents user programs from changing the registers' contents.

The operating system, executing in monitor mode, is given unrestricted access to both monitor and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump them out in case of errors, to access and modify parameters of system calls, and so on.

2.5.4 CPU Protection

The third piece of the protection puzzle is ensuring that the operating system maintains control. We must prevent a user program from getting stuck in an infinite loop, and never returning control to the operating system. To accomplish this goal, we can use a *timer*. A timer can be set to

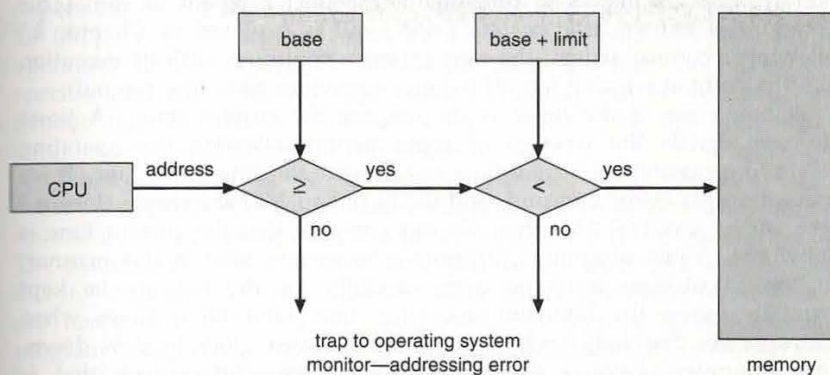


Figure 2.8 Hardware address protection with base and limit registers.

interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second, in increments of 1 millisecond). A variable timer is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches zero, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock would allow interrupts at intervals from 1 millisecond to 1024 milliseconds, in steps of 1 millisecond.

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

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

A more common use of a timer is to implement time sharing. In the most straightforward case, the timer could be set to interrupt every N milliseconds where N is the *time-slice* each user is allowed to execute before the next user gets control of the CPU. The operating system is invoked at the end of each time-slice to perform various housekeeping tasks, such as adding the value N to the record that specifies (for accounting purposes) the amount of time the user program has executed thus far. The operating system also resets registers, internal variables, and buffers, and changes several other parameters to prepare for the next program to run. (This procedure is known as a *context switch*, and is explored in Chapter 4.) Following a context switch, the next program continues with its execution from the point at which it left off (when its previous time-slice ran out).

Another use of the timer is to compute the current time. A timer interrupt signals the passage of some period, allowing the operating system to compute the current time in reference to some initial time. If we have interrupts every 1 second, and we have had 1427 interrupts since we were told it was 1:00 P.M., then we can compute that the current time is 1:23:47 P.M. Some computers determine the current time in this manner, but the calculations must be done carefully for the time to be kept accurately, since the interrupt-processing time (and other times when interrupts are disabled) tends to cause the software clock to slow down. Most computers have a separate hardware time-of-day clock that is independent of the operating system.

2.6 ■ General-System Architecture

The desire to improve the utilization of the computer system led to the development of multiprogramming and time sharing, where the resources of the computer system are shared among many different programs and processes. Sharing led directly to modifications of the basic computer architecture, to allow the operating system to maintain control over the computer system, and especially over I/O. Control must be maintained if we are to provide continuous, consistent, and correct operation.

To maintain control, developers introduced a dual mode of execution (user mode and monitor mode). This scheme supports the concept of privileged instructions, which can be executed only in monitor mode. I/O instructions and instructions to modify the memory-management registers or the timer are privileged instructions.

As you can imagine, several other instructions are also classified as privileged. For instance, the **halt** instruction is privileged; a user program should never be able to halt the computer. The instructions to turn the interrupt system on and off are also privileged, since proper operation of the timer and I/O depends on the ability to respond to interrupts correctly. The instruction to change from user mode to monitor mode is privileged, and on many machines any change to the mode bit is privileged.

Because I/O instructions are privileged, they can be executed by only the operating system. Then how does the user program perform I/O? By making I/O instructions privileged, we have prevented user programs from doing any I/O, either valid or invalid. The solution to this problem is that, because only the monitor can do I/O, the user must *ask* the monitor to do I/O on the user's behalf.

Such a request is known as a *system call* (also called a *monitor call* or *operating system function call*). A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic **trap** instruction, although some systems (like the Mips R2000 family) have a specific **syscall** instruction.

When a system call is executed, it is treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to monitor mode. The system-call service routine is a part of the operating system. The monitor examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The monitor verifies that the

parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

Thus, to do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf (Figure 2.9). The operating system, executing in monitor mode, checks that the request is valid, and (if the request is valid) does the I/O requested. The operating system then returns to the user.

2.7 ■ Summary

Multiprogramming and time-sharing systems require the overlap of CPU and I/O operations on a single machine. Such an overlap requires that data transfer between the CPU and the I/O devices be handled by one or both of the following methods: (1) interrupt-initialized data transfer, (2) DMA data transfer.

For a computer to do its job of executing programs, the programs must be in main memory. Main memory is the only large storage area that the processor can access directly. It is an array of words or bytes, ranging in

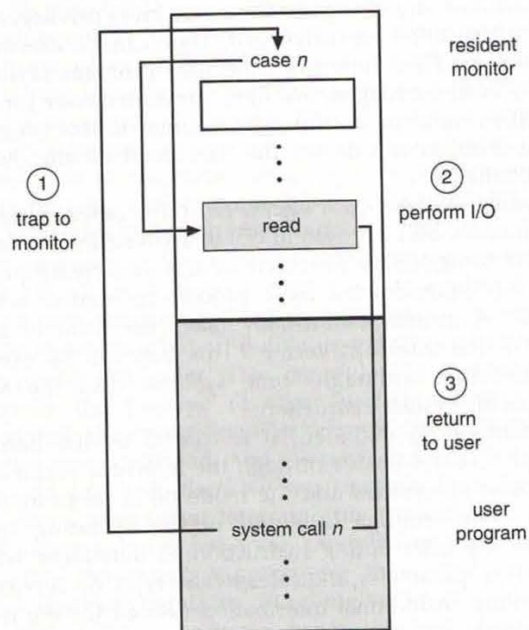


Figure 2.9 Use of a system call to perform I/O.

size from hundreds of thousands to hundreds of millions. Each word has its own address. The main memory is a *volatile* storage device that loses its contents when power is turned off or lost. Most computer systems provide *secondary storage* as an extension of main memory. The main requirement of secondary storage is to be able to hold extremely large numbers of data permanently. The most common secondary-storage device is a magnetic disk, which provides storage of both programs and data. A magnetic disk is a *nonvolatile* storage device that also provides random access. Magnetic tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

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

The operating system must ensure correct operation of the computer system. To prevent user programs from interfering with the proper operation of the system, developers modified the hardware to create two modes: user mode and monitor mode. Various instructions (such as I/O instructions and halt instructions) are privileged, and can be executed in only monitor mode. The memory in which the operating system resides must also be protected from modification by the user. A timer prevents infinite loops. Once these changes (dual mode, privileged instructions, memory protection, timer interrupt) have been made to the basic computer architecture, it is possible to ensure the correct operation of the system. Chapter 3 continues this discussion with details of the facilities that operating systems provide.

■ Exercises

- 2.1 Buffering is a method of overlapping the I/O of a job with that job's own computation. The idea is quite simple. After data have been read and the CPU is about to start operating on them, the input device is instructed to begin the next input immediately. The CPU and input device are then both busy. With luck, by the time that the CPU is ready for the next data item, the input device will have finished reading it. The CPU can then begin processing the newly read data, while the input device starts to read the following data. Similarly, the same process can be used for output. In this case, the CPU creates data that are put into a buffer until an output device can accept them.

Compare the buffering scheme with the spooling scheme where the CPU overlaps the input of one job with the computation and output of other jobs.

- 2.2 Show how a desire for control cards leads naturally to the creation of separate user and monitor modes of operation.
- 2.3 How does the distinction between monitor mode and user mode function as a rudimentary form of protection (security) system?
- 2.4 What are the differences between a trap and an interrupt? What is the use of each function?
- 2.5 For what types of operations is DMA useful? Why?
- 2.6 Which of the following instructions should be privileged?
 - a. Set value of timer.
 - b. Read the clock.
 - c. Clear memory.
 - d. Turn off interrupts.
 - e. Switch from user to monitor mode.
- 2.7 Many computer systems do not provide dual-mode operation in hardware. Consider whether it is possible to construct a secure operating system for these computers. Give arguments both that it is and that it is not possible.
- 2.8 Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme.
- 2.9 Protecting the operating system is crucial to ensuring that the computer system operates correctly. Provision of this protection is the reason behind dual-mode operation, memory protection, and the timer. To allow maximum flexibility, however, we would also like to place minimal constraints on the user. The following is a list of operations that are normally protected. What is the *minimal* set of instructions that must be protected?
 - a. Change to user mode.
 - b. Change to monitor mode.
 - c. Read from monitor memory.
 - d. Write into monitor memory.
 - e. Fetch an instruction from monitor memory.
 - f. Turn on timer interrupt.
 - g. Turn off timer interrupt.

- 2.10 When are caches useful? What problems do they solve? What problems do they cause? If a cache can be made as large as the device it is caching for (for instance, a cache as large as a disk) why not do so and eliminate the device?
- 2.11 Writing an operating system that can operate without interference from malicious or undebugged user programs requires some hardware assistance. Name three hardware aids for writing an operating system, and describe how they may be used together to protect the operating system.

Bibliographic Notes

A detailed description of I/O architectures such as channels and DMA on large systems appears in Baer [1980]. Hennessy and Patterson [1990] provide modern coverage of I/O systems and buses, and of system architecture in general. Tanenbaum [1990] describes the architecture of microcomputers, starting at a detailed hardware level.

General discussions concerning multiprocessing are given by Jones and Schwarz [1980]. Multiprocessor hardware is discussed by Satyanarayanan [1980]. Performance of multiprocessor systems is presented by Maples [1985], Sanguinetti [1986], Agrawal et al. [1986], and Bhuyan et al. [1989]. A survey of parallel computer architectures is presented by Duncan [1990].

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

Cache memories, including associative memory, are described and analyzed by Smith [1982]. This paper also includes an extensive bibliography on the subject. Hennessy and Patterson [1990] discuss the hardware aspects of TLBs, caches, and MMUs.

General discussions concerning mass-storage technology are offered by Chi [1982] and Hoagland [1985].