PART FIVE

Input/Output and Files

Perhaps the messiest parts of the design of an operating system deal with the I/O facility and the file management system. With respect to I/O, the key issue is performance. The I/O facility is truly the performance battleground. Looking at the internal operation of a computer system, we see that processor speed continues to increase and, if a single processor is still not fast enough, SMP configurations provide multiple processors to speed the work. Internal memory access speeds are also increasing, though not at as fast a rate as processor speed. Nevertheless, with the clever use of one, two, or even more levels of internal cache, main memory access time is managing to keep up with processor speed. But I/O remains a significant performance challenge, particularly in the case of disk storage.

With file systems, performance is also an issue. Other design requirements, such as reliability and security, also come into play. From a user's point of view, the file system is perhaps the most important aspect of the operating system: The user wants rapid access to files but also guarantees that the files will not be corrupted and that they are secure from unauthorized access.

ROAD MAP FOR PART FIVE

Chapter 11 I/O Management and Disk Scheduling

Chapter 11 begins with an overview of I/O storage devices and the organization of the I/O function within the operating system. This is followed by discussion of various buffering strategies to improve performance. The remainder of the chapter is devoted to disk I/O. We look at the way in which multiple disk requests can be scheduled to take advantage of the physical characteristics of disk access to improve response time. Then we examine the use of a disk array to improve performance and reliability. Finally, we discuss the disk cache.

Chapter 12 File Management

Chapter 12 provides a survey of various types of file organizations and examines operating system issues related to file management and file access. It discusses physical and logical organization of data. It examines the services relating to file management that a typical operating system provides for users. It then looks at the specific mechanisms and data structures that are part of a file management system.

CHAPTER 11

I/O Management and Disk Scheduling

Perhaps the messiest aspect of operating system design is input/output. Because there is such a wide variety of devices and applications of those devices, it is difficult to develop a general, consistent solution.

We begin this chapter with a brief discussion of I/O devices and the organization of the I/O functions. These topics, which generally come within the scope of computer architecture, set the stage for an examination of I/O from the point of view of the operating system.

The next section examines operating system design issues, including design objectives, and the way in which the I/O function can be structured. Then I/O buffering is examined; one of the basic I/O services provided by the operating system is a buffering function, which improves overall performance.

The next sections of the chapter are devoted to magnetic disk I/O. In contemporary systems, this form of I/O is the most important and is key to the performance as perceived by the user. We begin by developing a model of disk I/O performance and then examine several techniques that can be used to enhance performance.

An appendix to this chapter summarizes characteristics of secondary storage devices, including magnetic disk and optical memory.

11.1 I/O DEVICES

As was mentioned in Chapter 1, external devices that engage in I/O with computer systems can be roughly grouped into three categories:

- **Human readable:** Suitable for communicating with the computer user. Examples include printers and terminals, the latter consisting of video display, keyboard, and perhaps other devices such as a mouse.
- **Machine readable:** Suitable for communicating with electronic equipment. Examples are disk drives, USB keys, sensors, controllers, and actuators.
- **Communication:** Suitable for communicating with remote devices. Examples are digital line drivers and modems.

There are great differences across classes and even substantial differences within each class. Among the key differences are the following:

- **Data rate:** There may be differences of several orders of magnitude between the data transfer rates. Figure 11.1 gives some examples.
- **Application:** The use to which a device is put has an influence on the software and policies in the operating system and supporting utilities. For example, a disk used for files requires the support of file management software. A disk used as a backing store for pages in a virtual memory scheme depends on the use of virtual memory hardware and software. Furthermore, these applications have an impact on disk scheduling algorithms (discussed later in this chapter). As another example, a terminal may be used by an ordinary user or a system administrator. These uses imply different privilege levels and perhaps different priorities in the operating system.
- **Complexity of control:** A printer requires a relatively simple control interface. A disk is much more complex. The effect of these differences on the operating

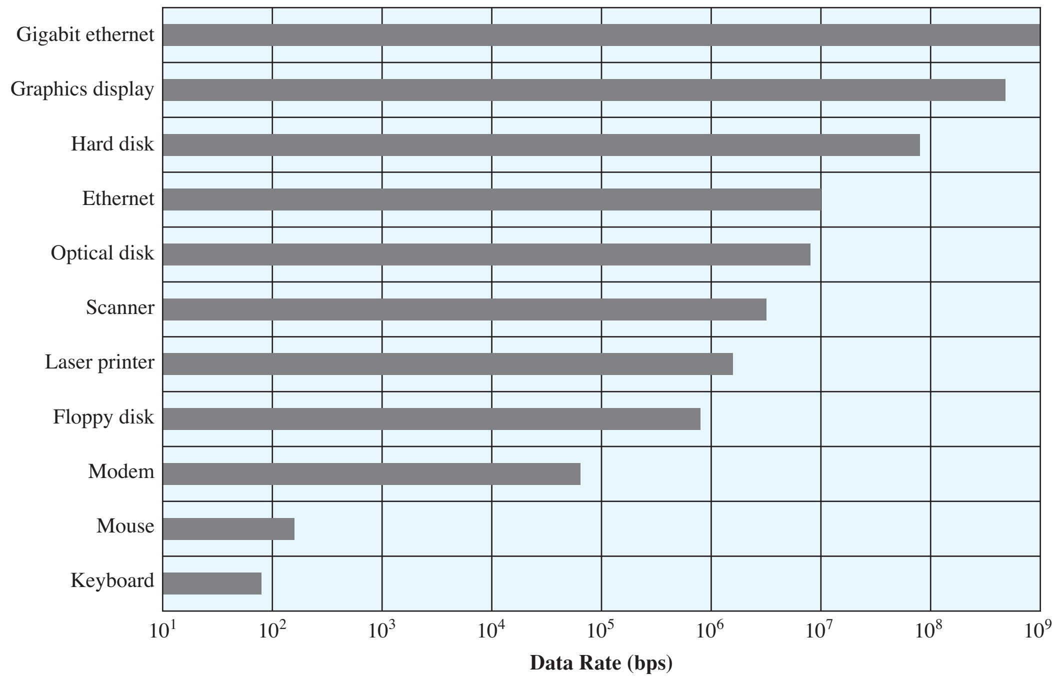


Figure 11.1 **Typical I/O Device Data Rates**

system is filtered to some extent by the complexity of the I/O module that controls the device, as discussed in the next section.

- **Unit of transfer:** Data may be transferred as a stream of bytes or characters (e.g., terminal I/O) or in larger blocks (e.g., disk I/O).
- **Data representation:** Different data encoding schemes are used by different devices, including differences in character code and parity conventions.
- **Error conditions:** The nature of errors, the way in which they are reported, their consequences, and the available range of responses differ widely from one device to another.

This diversity makes a uniform and consistent approach to I/O, both from the point of view of the operating system and from the point of view of user processes, difficult to achieve.

11.2 ORGANIZATION OF THE I/O FUNCTION

Section 1.7 summarized three techniques for performing I/O:

- **Programmed I/O:** The processor issues an I/O command, on behalf of a process, to an I/O module; that process then busy waits for the operation to be completed before proceeding.

Table 11.1 I/O Techniques

	No Interrupts	**Use of Interrupts**
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

- **Interrupt-driven I/O:** The processor issues an I/O command on behalf of a process. There are then two possibilities. If the I/O instruction from the process is nonblocking, then the processor continues to execute instructions from the process that issued the I/O command. If the I/O instruction is blocking, then the next instruction that the processor executes is from the OS, which will put the current process in a blocked state and schedule another process.
- **Direct memory access (DMA):** A DMA module controls the exchange of data between main memory and an I/O module. The processor sends a request for the transfer of a block of data to the DMA module and is interrupted only after the entire block has been transferred.

Table 11.1 indicates the relationship among these three techniques. In most computer systems, DMA is the dominant form of transfer that must be supported by the operating system.

The Evolution of the I/O Function

As computer systems have evolved, there has been a pattern of increasing complexity and sophistication of individual components. Nowhere is this more evident than in the I/O function. The evolutionary steps can be summarized as follows:

1. The processor directly controls a peripheral device. This is seen in simple microprocessor-controlled devices.
2. A controller or I/O module is added. The processor uses programmed I/O without interrupts. With this step, the processor becomes somewhat divorced from the specific details of external device interfaces.
3. The same configuration as step 2 is used, but now interrupts are employed. The processor need not spend time waiting for an I/O operation to be performed, thus increasing efficiency.
4. The I/O module is given direct control of memory via DMA. It can now move a block of data to or from memory without involving the processor, except at the beginning and end of the transfer.
5. The I/O module is enhanced to become a separate processor, with a specialized instruction set tailored for I/O. The central processing unit (CPU) directs the I/O processor to execute an I/O program in main memory. The I/O processor fetches and executes these instructions without processor intervention. This allows the processor to specify a sequence of I/O activities and to be interrupted only when the entire sequence has been performed.
6. The I/O module has a local memory of its own and is, in fact, a computer in its own right. With this architecture, a large set of I/O devices can be controlled,

with minimal processor involvement. A common use for such an architecture has been to control communications with interactive terminals. The I/O processor takes care of most of the tasks involved in controlling the terminals.

As one proceeds along this evolutionary path, more and more of the I/O function is performed without processor involvement. The central processor is increasingly relieved of I/O-related tasks, improving performance. With the last two steps (5 and 6), a major change occurs with the introduction of the concept of an I/O module capable of executing a program.

A note about terminology: For all of the modules described in steps 4 through 6, the term *direct memory access* is appropriate, because all of these types involve direct control of main memory by the I/O module. Also, the I/O module in step 5 is often referred to as an **I/O channel**, and that in step 6 as an **I/O processor**; however, each term is, on occasion, applied to both situations. In the latter part of this section, we will use the term *I/O channel* to refer to both types of I/O modules.

Direct Memory Access

Figure 11.2 indicates, in general terms, the DMA logic. The DMA unit is capable of mimicking the processor and, indeed, of taking over control of the system bus just like a processor. It needs to do this to transfer data to and from memory over the system bus.

The DMA technique works as follows. When the processor wishes to read or write a block of data, it issues a command to the DMA module by sending to the DMA module the following information:

- Whether a read or write is requested, using the read or write control line between the processor and the DMA module

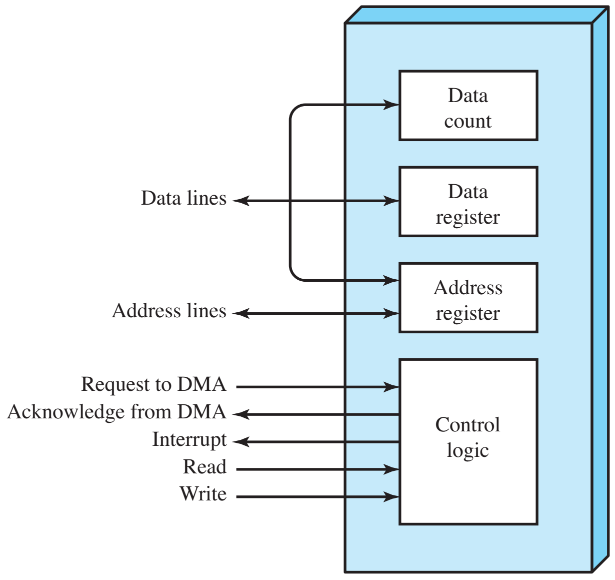


Figure 11.2 Typical DMA Block Diagram

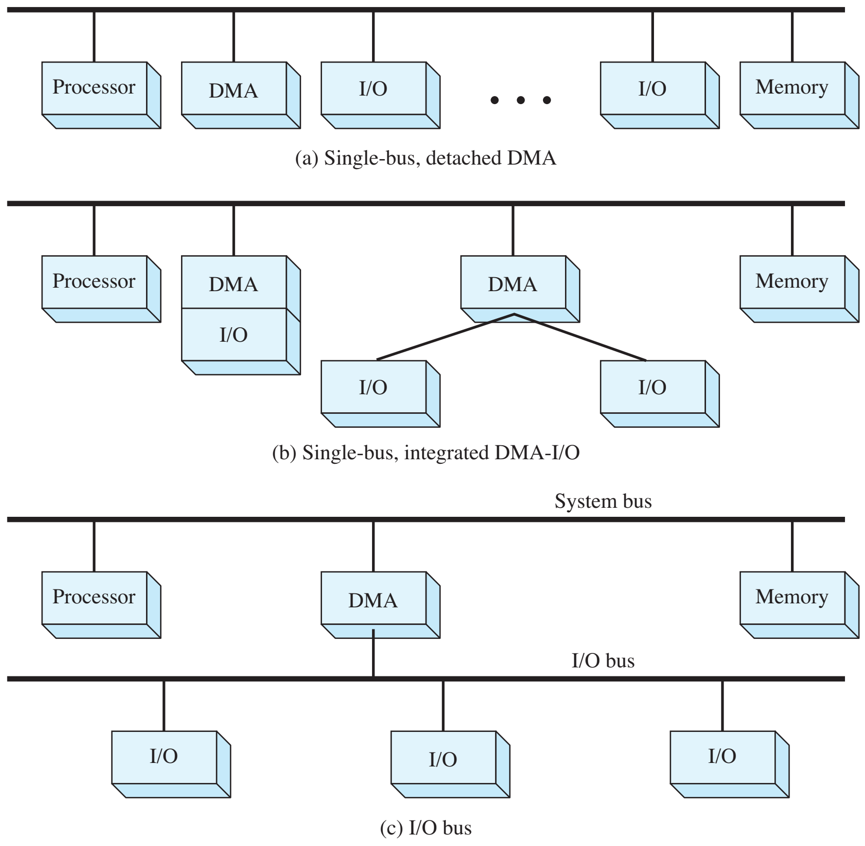


Figure 11.3 Alternative DMA Configurations

- The address of the I/O device involved, communicated on the data lines
- The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its address register
- The number of words to be read or written, again communicated via the data lines and stored in the data count register

The processor then continues with other work. It has delegated this I/O operation to the DMA module. The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, the processor is involved only at the beginning and end of the transfer (Figure 1.19c).

The DMA mechanism can be configured in a variety of ways. Some possibilities are shown in Figure 11.3. In the first example, all modules share the same system bus. The DMA module, acting as a surrogate processor, uses programmed I/O to exchange data between memory and an I/O module through the DMA module. This configuration, while it may be inexpensive, is clearly inefficient: As with processor-controlled programmed I/O, each transfer of a word consumes two bus cycles (transfer request followed by transfer).

The number of required bus cycles can be cut substantially by integrating the DMA and I/O functions. As Figure 11.3b indicates, this means that there is a path between the DMA module and one or more I/O modules that does not include the system bus. The DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules. This concept can be taken one step further by connecting I/O modules to the DMA module using an I/O bus (Figure 11.3c). This reduces the number of I/O interfaces in the DMA module to one and provides for an easily expandable configuration. In all of these cases (Figure 11.3b and c), the system bus that the DMA module shares with the processor and main memory is used by the DMA module only to exchange data with memory and to exchange control signals with the processor. The exchange of data between the DMA and I/O modules takes place off the system bus.

11.3 OPERATING SYSTEM DESIGN ISSUES

Design Objectives

Two objectives are paramount in designing the I/O facility: efficiency and generality. **Efficiency** is important because I/O operations often form a bottleneck in a computing system. Looking again at Figure 11.1, we see that most I/O devices are extremely slow compared with main memory and the processor. One way to tackle this problem is multiprogramming, which, as we have seen, allows some processes to be waiting on I/O operations while another process is executing. However, even with the vast size of main memory in today's machines, it will still often be the case that I/O is not keeping up with the activities of the processor. Swapping is used to bring in additional ready processes to keep the processor busy, but this in itself is an I/O operation. Thus, a major effort in I/O design has been schemes for improving the efficiency of the I/O. The area that has received the most attention, because of its importance, is disk I/O, and much of this chapter will be devoted to a study of disk I/O efficiency.

The other major objective is **generality**. In the interests of simplicity and freedom from error, it is desirable to handle all devices in a uniform manner. This statement applies both to the way in which processes view I/O devices and the way in which the operating system manages I/O devices and operations. Because of the diversity of device characteristics, it is difficult in practice to achieve true generality. What can be done is to use a hierarchical, modular approach to the design of the I/O function. This approach hides most of the details of device I/O in lower-level routines so that user processes and upper levels of the operating system see devices in terms of general functions, such as read, write, open, close, lock, unlock. We turn now to a discussion of this approach.

Logical Structure of the I/O Function

In Chapter 2, in the discussion of system structure, we emphasized the hierarchical nature of modern operating systems. The hierarchical philosophy is that the functions of the operating system should be separated according to their complexity, their characteristic time scale, and their level of abstraction. Following this approach leads to an organization of the operating system into a series of layers. Each layer performs a related subset of the functions required of the operating system. It relies on the next lower layer to perform more primitive functions and to conceal the details of those functions. It provides services to the next higher layer. Ideally, the layers should be defined so that changes in one layer do not require changes in other layers. Thus we have decomposed one problem into a number of more manageable subproblems.

In general, lower layers deal with a far shorter time scale. Some parts of the operating system must interact directly with the computer hardware, where events can have a time scale as brief as a few billionths of a second. At the other end of the spectrum, parts of the operating system communicate with the user, who issues commands at a much more leisurely pace, perhaps one every few seconds. The use of a set of layers conforms nicely to this environment.

Applying this philosophy specifically to the I/O facility leads to the type of organization suggested by Figure 11.4 (compare with Table 2.4). The details of the organization will depend on the type of device and the application. The three most

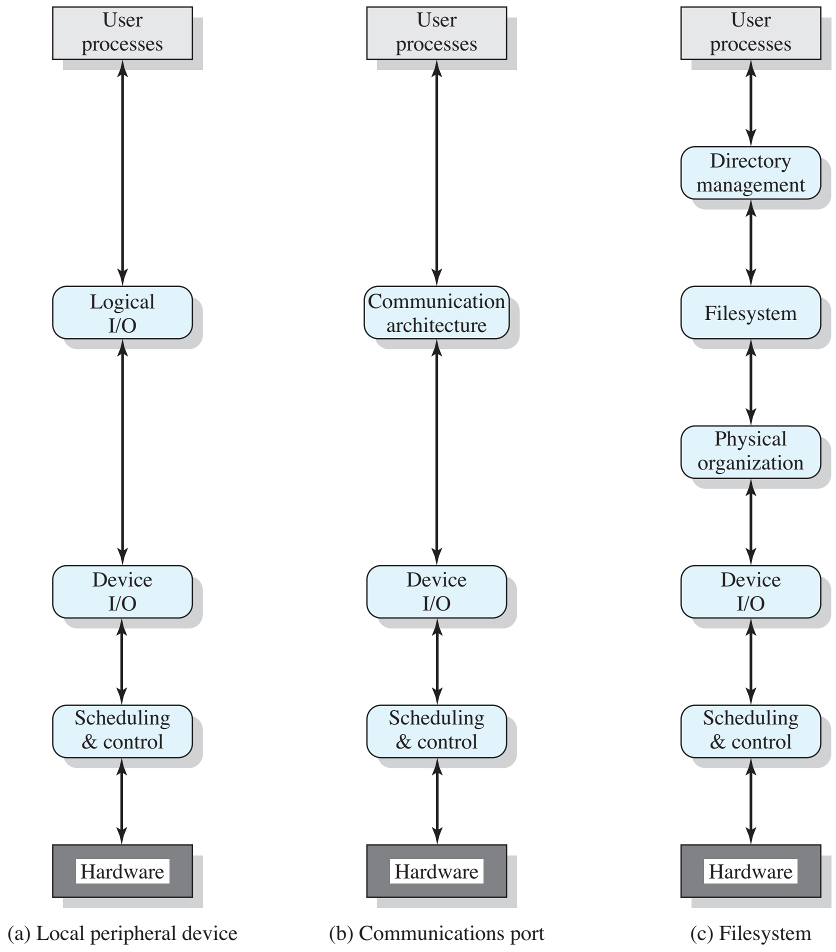


Figure 11.4 A Model of I/O Organization

important logical structures are presented in the figure. Of course, a particular operating system may not conform exactly to these structures. However, the general principles are valid, and most operating systems approach I/O in approximately this way.

Let us consider the simplest case first, that of a local peripheral device that communicates in a simple fashion, such as a stream of bytes or records (Figure 11.4a). The following layers are involved:

- **Logical I/O:** The logical I/O module deals with the device as a logical resource and is not concerned with the details of actually controlling the device. The logical I/O module is concerned with managing general I/O functions on behalf of user processes, allowing them to deal with the device in terms of a device identifier and simple commands such as open, close, read, write.
- **Device I/O:** The requested operations and data (buffered characters, records, etc.) are converted into appropriate sequences of I/O instructions, channel commands, and controller orders. Buffering techniques may be used to improve utilization.
- **Scheduling and control:** The actual queuing and scheduling of I/O operations occurs at this layer, as well as the control of the operations. Thus, interrupts are handled at this layer and I/O status is collected and reported. This is the layer of software that actually interacts with the I/O module and hence the device hardware.

For a communications device, the I/O structure (Figure 11.4b) looks much the same as that just described. The principal difference is that the logical I/O module is replaced by a communications architecture, which may itself consist of a number of layers. An example is TCP/IP, which is discussed in Chapter 17.

Figure 11.4c shows a representative structure for managing I/O on a secondary storage device that supports a file system. The three layers not previously discussed are as follows:

- **Directory management:** At this layer, symbolic file names are converted to identifiers that either reference the file directly or indirectly through a file descriptor or index table. This layer is also concerned with user operations that affect the directory of files, such as add, delete, and reorganize.
- **File system:** This layer deals with the logical structure of files and with the operations that can be specified by users, such as open, close, read, write. Access rights are also managed at this layer.
- **Physical organization:** Just as virtual memory addresses must be converted into physical main memory addresses, taking into account the segmentation and paging structure, logical references to files and records must be converted to physical secondary storage addresses, taking into account the physical track and sector structure of the secondary storage device. Allocation of secondary storage space and main storage buffers is generally treated at this layer as well.

Because of the importance of the file system, we will spend some time, in this chapter and the next, looking at its various components. The discussion in this chapter focuses on the lower three layers, while the upper two layers are examined in Chapter 12.

11.4 I/O BUFFERING

Suppose that a user process wishes to read blocks of data from a disk one at a time, with each block having a length of 512 bytes. The data are to be read into a data area within the address space of the user process at virtual location 1000 to 1511 The simplest way would be to execute an I/O command (something like Read_Block[1000, disk]) to the disk unit and then wait for the data to become available. The waiting could either be busy waiting (continuously test the device status) or, more practically, process suspension on an interrupt.

There are two problems with this approach. First, the program is hung up waiting for the relatively slow I/O to complete. The second problem is that this approach to I/O interferes with swapping decisions by the operating system. Virtual locations 1000 to 1511 must remain in main memory during the course of the block transfer. Otherwise, some of the data may be lost. If paging is being used, at least the page containing the target locations must be locked into main memory. Thus, although portions of the process may be paged out to disk, it is impossible to swap the process out completely, even if this is desired by the operating system. Notice also that there is a risk of single-process deadlock. If a process issues an I/O command, is suspended awaiting the result, and then is swapped out prior to the beginning of the operation, the process is blocked waiting on the I/O event, and the I/O operation is blocked waiting for the process to be swapped in. To avoid this deadlock, the user memory involved in the I/O operation must be locked in main memory immediately before the I/O request is issued, even though the I/O operation is queued and may not be executed for some time.

The same considerations apply to an output operation. If a block is being transferred from a user process area directly to an I/O module, then the process is blocked during the transfer and the process may not be swapped out.

To avoid these overheads and inefficiencies, it is sometimes convenient to perform input transfers in advance of requests being made and to perform output transfers some time after the request is made. This technique is known as buffering. In this section, we look at some of the buffering schemes that are supported by operating systems to improve the performance of the system.

In discussing the various approaches to buffering, it is sometimes important to make a distinction between two types of I/O devices: block oriented and stream oriented. A **block-oriented** device stores information in blocks that are usually of fixed size, and transfers are made one block at a time. Generally, it is possible to reference data by its block number. Disks and USB keys are examples of block-oriented devices. A **stream-oriented** device transfers data in and out as a stream of bytes, with no block structure. Terminals, printers, communications ports, mouse and other pointing devices, and most other devices that are not secondary storage are stream oriented.

Single Buffer

The simplest type of support that the operating system can provide is single buffering (Figure 11.5b). When a user process issues an I/O request, the operating system assigns a buffer in the system portion of main memory to the operation.

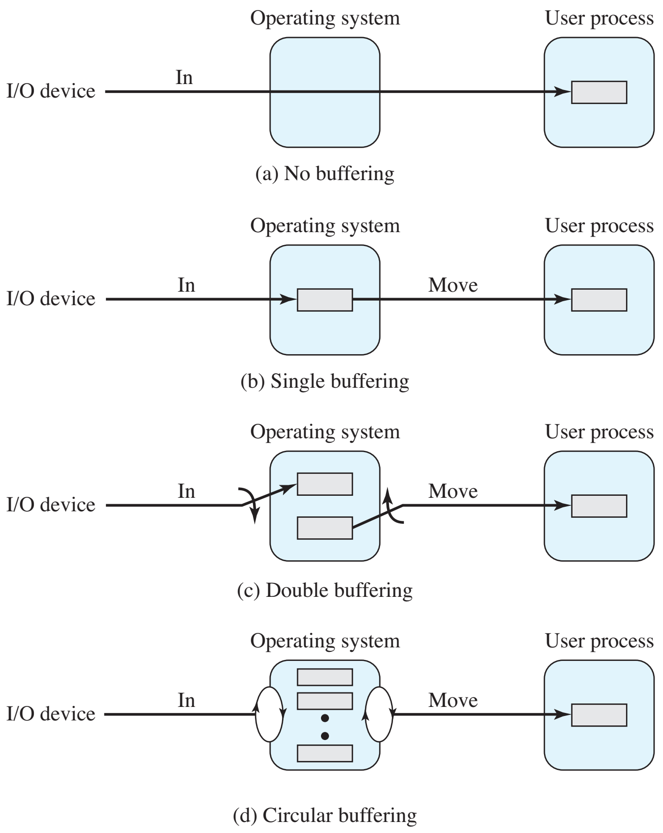


Figure 11.5 I/O Buffering Schemes (input)

For block-oriented devices, the single buffering scheme can be described as follows: Input transfers are made to the system buffer. When the transfer is complete, the process moves the block into user space and immediately requests another block. This is called reading ahead, or anticipated input; it is done in the expectation that the block will eventually be needed. For many types of computation, this is a reasonable assumption most of the time because data are usually accessed sequentially. Only at the end of a sequence of processing will a block be read in unnecessarily.

This approach will generally provide a speedup compared to the lack of system buffering. The user process can be processing one block of data while the next block is being read in. The operating system is able to swap the process out because the input operation is taking place in system memory rather than user process memory. This technique does, however, complicate the logic in the operating system. The operating system must keep track of the assignment of system buffers to user processes. The swapping logic is also affected: If the I/O operation involves the same disk that is used for swapping, it hardly makes sense to queue disk writes to the same device for swapping the process out. This attempt to swap the process and release main memory will itself not begin until after the I/O operation finishes, at which time swapping the process to disk may no longer be appropriate.

Similar considerations apply to block-oriented output. When data are being transmitted to a device, they are first copied from the user space into the system buffer, from which they will ultimately be written. The requesting process is now free to continue or to be swapped as necessary.

[KNUT97] suggests a crude but informative performance comparison between single buffering and no buffering. Suppose that T is the time required to input one block and that C is the computation time that intervenes between input requests. Without buffering, the execution time per block is essentially $T + C$. With a single buffer, the time is max $[C, T] + M$, where M is the time required to move the data from the system buffer to user memory. In most cases, execution time per block is substantially less with a single buffer compared to no buffer.

For stream-oriented I/O, the single buffering scheme can be used in a line-at-a-time fashion or a byte-at-a-time fashion. Line-at-a-time operation is appropriate for scroll-mode terminals (sometimes called dumb terminals). With this form of terminal, user input is one line at a time, with a carriage return signaling the end of a line, and output to the terminal is similarly one line at a time. A line printer is another example of such a device. Byte-at-a-time operation is used on forms-mode terminals, when each keystroke is significant, and for many other peripherals, such as sensors and controllers.

In the case of line-at-a-time I/O, the buffer can be used to hold a single line. The user process is suspended during input, awaiting the arrival of the entire line. For output, the user process can place a line of output in the buffer and continue processing. It need not be suspended unless it has a second line of output to send before the buffer is emptied from the first output operation. In the case of byte-at-a-time I/O, the interaction between the operating system and the user process follows the producer/consumer model discussed in Chapter 5.

Double Buffer

An improvement over single buffering can be had by assigning two system buffers to the operation (Figure 11.5c). A process now transfers data to (or from) one buffer while the operating system empties (or fills) the other. This technique is known as **double buffering** or **buffer swapping**.

For block-oriented transfer, we can roughly estimate the execution time as max $[C, T]$. It is therefore possible to keep the block-oriented device going at full speed if $C \leq T$. On the other hand, if $C > T$, double buffering ensures that the process will not have to wait on I/O. In either case, an improvement over single buffering is achieved. Again, this improvement comes at the cost of increased complexity.

For stream-oriented input, we again are faced with the two alternative modes of operation. For line-at-a-time I/O, the user process need not be suspended for input or output, unless the process runs ahead of the double buffers. For byte-at-a-time operation, the double buffer offers no particular advantage over a single buffer of twice the length. In both cases, the producer/consumer model is followed.

Circular Buffer

A double-buffer scheme should smooth out the flow of data between an I/O device and a process. If the performance of a particular process is the focus of our concern, then we would like for the I/O operation to be able to keep up with the process. Double buffering may be inadequate if the process performs rapid bursts of I/O. In this case, the problem can often be alleviated by using more than two buffers.

When more than two buffers are used, the collection of buffers is itself referred to as a circular buffer (Figure 11.5d), with each individual buffer being one unit in the circular buffer. This is simply the bounded-buffer producer/consumer model studied in Chapter 5.

The Utility of Buffering

Buffering is a technique that smoothes out peaks in I/O demand. However, no amount of buffering will allow an I/O device to keep pace with a process indefinitely when the average demand of the process is greater than the I/O device can service. Even with multiple buffers, all of the buffers will eventually fill up and the process will have to wait after processing each chunk of data. However, in a multiprogramming environment, when there is a variety of I/O activity and a variety of process activity to service, buffering is one tool that can increase the efficiency of the operating system and the performance of individual processes.

11.5 DISK SCHEDULING

Over the last 40 years, the increase in the speed of processors and main memory has far outstripped that for disk access, with processor and main memory speeds increasing by about two orders of magnitude compared to one order of magnitude for disk. The result is that disks are currently at least four orders of magnitude slower than main memory. This gap is expected to continue into the foreseeable future. Thus, the performance of disk storage subsystem is of vital concern, and much research has gone into schemes for improving that performance. In this section, we highlight some of the key issues and look at the most important approaches. Because the performance of the disk system is tied closely to file system design issues, the discussion continues in Chapter 12.

Disk Performance Parameters

The actual details of disk I/O operation depend on the computer system, the operating system, and the nature of the I/O channel and disk controller hardware. A general timing diagram of disk I/O transfer is shown in Figure 11.6.

When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track.[1] Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system. On a movable-head system, the time it takes to position the head at the track is known as **seek time**. In either case, once the track is selected, the disk controller waits until the appropriate sector rotates to line up with the head. The time it takes for the beginning

[1]See Appendix 11A for a discussion of disk organization and formatting.

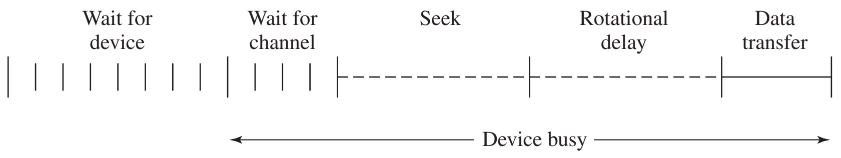


Figure 11.6 Timing of a Disk I/O Transfer

of the sector to reach the head is known as **rotational delay**, or rotational latency. The sum of the seek time, if any, and the rotational delay equals the **access time**, which is the time it takes to get into position to read or write. Once the head is in position, the read or write operation is then performed as the sector moves under the head; this is the data transfer portion of the operation; the time required for the transfer is the **transfer time**.

In addition to the access time and transfer time, there are several queuing delays normally associated with a disk I/O operation. When a process issues an I/O request, it must first wait in a queue for the device to be available. At that time, the device is assigned to the process. If the device shares a single I/O channel or a set of I/O channels with other disk drives, then there may be an additional wait for the channel to be available. At that point, the seek is performed to begin disk access.

In some high-end systems for servers, a technique known as rotational positional sensing (RPS) is used. This works as follows: When the seek command has been issued, the channel is released to handle other I/O operations. When the seek is completed, the device determines when the data will rotate under the head. As that sector approaches the head, the device tries to reestablish the communication path back to the host. If either the control unit or the channel is busy with another I/O, then the reconnection attempt fails and the device must rotate one whole revolution before it can attempt to reconnect, which is called an RPS miss. This is an extra delay element that must be added to the time line of Figure 11.6.

Seek Time Seek time is the time required to move the disk arm to the required track. It turns out that this is a difficult quantity to pin down. The seek time consists of two key components: the initial startup time and the time taken to traverse the tracks that have to be crossed once the access arm is up to speed. Unfortunately, the traversal time is not a linear function of the number of tracks but includes a settling time (time after positioning the head over the target track until track identification is confirmed).

Much improvement comes from smaller and lighter disk components. Some years ago, a typical disk was 14 inches (36 cm) in diameter, whereas the most common size today is 3.5 inches (8.9 cm), reducing the distance that the arm has to travel. A typical average seek time on contemporary hard disks is under 10 ms.

Rotational Delay Rotational delay is the time required for the addressed area of the disk to rotate into a position where it is accessible by the read/write head. Disks, other than floppy disks, rotate at speeds ranging from 3600 rpm (for handheld devices such as digital cameras) up to, as of this writing, 15,000 rpm; at this latter speed, there is one revolution per 4 ms. Thus, on the average, the rotational delay will be 2 ms. Floppy disks typically rotate at between 300 and 600 rpm. Thus the average delay will be between 100 and 50 ms.

Transfer Time The transfer time to or from the disk depends on the rotation speed of the disk in the following fashion:

$$T = \frac{b}{rN}$$

where

T = transfer time
b = number of bytes to be transferred
N = number of bytes on a track
r = rotation speed, in revolutions per second

Thus the total average access time can be expressed as

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

where T_s is the average seek time.

A Timing Comparison With the foregoing parameters defined, let us look at two different I/O operations that illustrate the danger of relying on average values. Consider a disk with an advertised average seek time of 4 ms, rotation speed of 7500 rpm, and 512-byte sectors with 500 sectors per track. Suppose that we wish to read a file consisting of 2500 sectors for a total of 1.28 Mbytes. We would like to estimate the total time for the transfer.

First, let us assume that the file is stored as compactly as possible on the disk. That is, the file occupies all of the sectors on 5 adjacent tracks (5 tracks × 500 sectors/track = 2500 sectors). This is known as *sequential organization*. The time to read the first track is as follows:

Average seek	4 ms
Rotational delay	4 ms
Read 500 sectors	8 ms
	16 ms

Suppose that the remaining tracks can now be read with essentially no seek time. That is, the I/O operation can keep up with the flow from the disk. Then, at most, we need to deal with rotational delay for each succeeding track. Thus, each successive track is read in 4 + 8 = 12 ms. To read the entire file;

$$\text{Total time} = 16 + (4 \times 12) = 64 \text{ ms} = 0.064 \text{ seconds}$$

Now let us calculate the time required to read the same data using random access rather than sequential access; that is, accesses to the sectors are distributed randomly over the disk. For each sector, we have

Average seek	4 ms
Rotational delay	4 ms
Read 1 sector	0.016 ms
	8.016 ms

$$\text{Total time} = 2500 \times 8.016 = 20{,}040 \text{ ms} = 20.04 \text{ seconds}$$

It is clear that the order in which sectors are read from the disk has a tremendous effect on I/O performance. In the case of file access in which multiple sectors are read or written, we have some control over the way in which sectors of data are deployed, and we shall have something to say on this subject in the next chapter. However, even in the case of a file access, in a multiprogramming environment, there will be I/O requests competing for the same disk. Thus, it is worthwhile to examine ways in which the performance of disk I/O can be improved over that achieved with purely random access to the disk.

Animation
Disk Scheduling
Algorithms

Disk Scheduling Policies

In the example just described, the reason for the difference in performance can be traced to seek time. If sector access requests involve selection of tracks at random, then the performance of the disk I/O system will be as poor as possible. To improve matters, we need to reduce the average time spent on seeks.

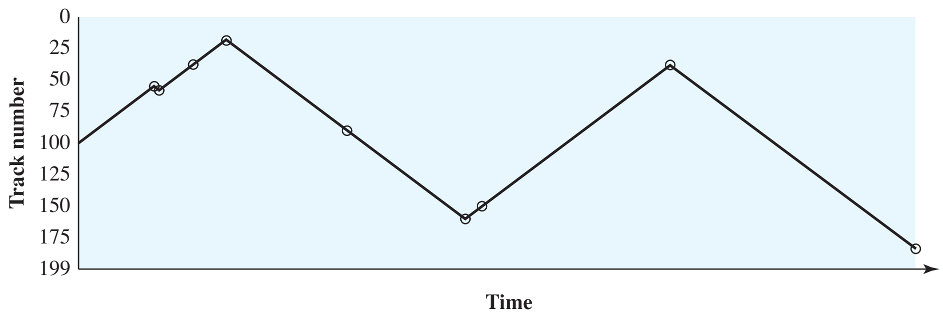
Consider the typical situation in a multiprogramming environment, in which the operating system maintains a queue of requests for each I/O device. So, for a single disk, there will be a number of I/O requests (reads and writes) from various processes in the queue. If we selected items from the queue in random order, then we can expect that the tracks to be visited will occur randomly, giving poor performance. This **random scheduling** is useful as a benchmark against which to evaluate other techniques.

Figure 11.7 compares the performance of various scheduling algorithms for an example sequence of I/O requests. The vertical axis corresponds to the tracks on the disk. The horizontal access corresponds to time or, equivalently, the number of tracks traversed. For this figure, we assume that the disk head is initially located at track 100. In this example, we assume a disk with 200 tracks and that the disk request queue has random requests in it. The requested tracks, in the order received by the disk scheduler, are 55, 58, 39, 18, 90, 160, 150, 38, 184. Table 11.2a tabulates the results.

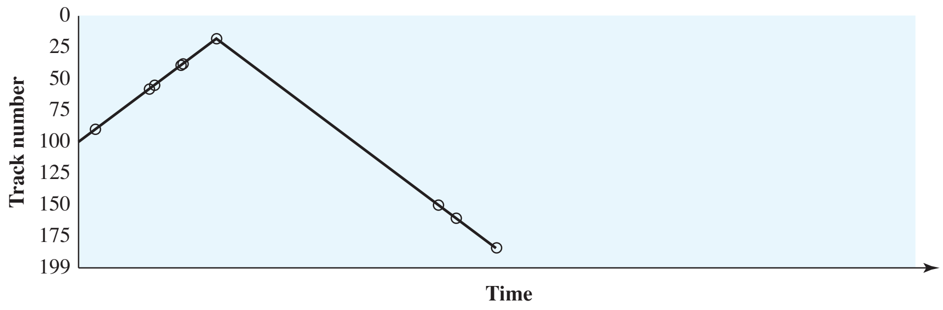
First-In-First-Out

The simplest form of scheduling is first-in-first-out (FIFO) scheduling, which processes items from the queue in sequential order. This strategy has the advantage of being fair, because every request is honored and the requests are honored in the order received. Figure 11.7a illustrates the disk arm movement with FIFO. This graph is generated directly from the data in Table 11.2a. As can be seen, the disk accesses are in the same order as the requests were originally received.

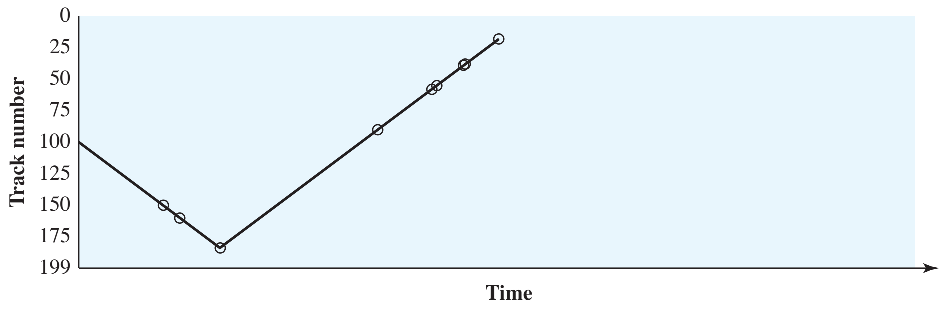
With FIFO, if there are only a few processes that require access and if many of the requests are to clustered file sectors, then we can hope for good performance. However, this technique will often approximate random scheduling in performance, if there are many processes competing for the disk. Thus, it may be profitable to consider a more sophisticated scheduling policy. A number of these are listed in Table 11.3 and will now be considered.



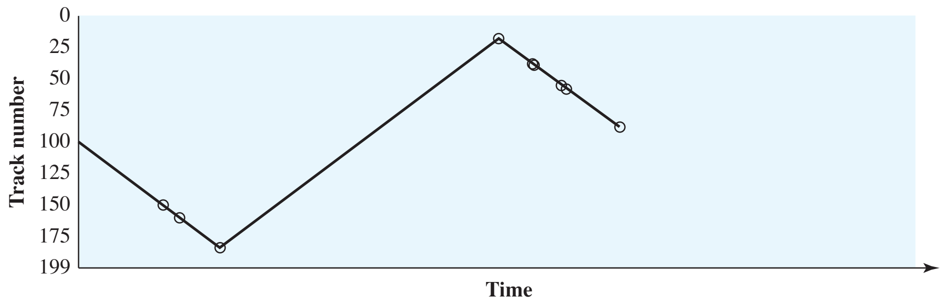
(a) FIFO



(b) SSTF



(c) SCAN



(d) C-SCAN

Figure 11.7 Comparison of Disk Scheduling Algorithms (see Table 11.3)

Table 11.2 Comparison of Disk Scheduling Algorithms

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	**Average seek length**	27.5	**Average seek length**	27.8	**Average seek length**	35.8

Priority With a system based on priority (PRI), the control of the scheduling is outside the control of disk management software. Such an approach is not intended to optimize disk utilization but to meet other objectives within the operating system. Often short batch jobs and interactive jobs are given higher priority than longer jobs that require longer computation. This allows a lot of short jobs to be flushed through the system quickly and may provide good interactive response time. However, longer jobs may have to wait excessively long times. Furthermore, such a policy could lead to

Table 11.3 Disk Scheduling Algorithms

Name	Description	Remarks
Selection according to requestor		
RSS	Random scheduling	For analysis and simulation
FIFO	First in first out	Fairest of them all
PRI	Priority by process	Control outside of disk queue management
LIFO	Last in first out	Maximize locality and resource utilization
Selection according to requested item		
SSTF	Shortest service time first	High utilization, small queues
SCAN	Back and forth over disk	Better service distribution
C-SCAN	One way with fast return	Lower service variability
N-step-SCAN	SCAN of *N* records at a time	Service guarantee
FSCAN	N-step-SCAN with *N* = queue size at beginning of SCAN cycle	Load sensitive

countermeasures on the part of users, who split their jobs into smaller pieces to beat the system. This type of policy tends to be poor for database systems.

Last In First Out Surprisingly, a policy of always taking the most recent request has some merit. In transaction processing systems, giving the device to the most recent user should result in little or no arm movement for moving through a sequential file. Taking advantage of this locality improves throughput and reduces queue lengths. As long as a job can actively use the file system, it is processed as fast as possible. However, if the disk is kept busy because of a large workload, there is the distinct possibility of starvation. Once a job has entered an I/O request in the queue and fallen back from the head of the line, the job can never regain the head of the line unless the queue in front of it empties.

FIFO, priority, and LIFO (last in first out) scheduling are based solely on attributes of the queue or the requester. If the scheduler knows the current track position, then scheduling based on the requested item can be employed. We examine these policies next.

Shortest Service Time First The SSTF policy is to select the disk I/O request that requires the least movement of the disk arm from its current position. Thus, we always choose to incur the minimum seek time. Of course, always choosing the minimum seek time does not guarantee that the average seek time over a number of arm movements will be minimum. However, this should provide better performance than FIFO. Because the arm can move in two directions, a random tie-breaking algorithm may be used to resolve cases of equal distances.

Figure 11.7b and Table 11.2b show the performance of SSTF on the same example as was used for FIFO. The first track accessed is 90, because this is the closest requested track to the starting position. The next track accessed is 58 because this is the closest of the remaining requested tracks to the current position of 90. Subsequent tracks are selected accordingly.

SCAN With the exception of FIFO, all of the policies described so far can leave some request unfulfilled until the entire queue is emptied. That is, there may always be new requests arriving that will be chosen before an existing request. A simple alternative that prevents this sort of starvation is the SCAN algorithm, also known as the elevator algorithm because it operates much the way an elevator does.

With SCAN, the arm is required to move in one direction only, satisfying all outstanding requests en route, until it reaches the last track in that direction or until there are no more requests in that direction. This latter refinement is sometimes referred to as the LOOK policy. The service direction is then reversed and the scan proceeds in the opposite direction, again picking up all requests in order.

Figure 11.7c and Table 11.2c illustrate the SCAN policy. Assuming that the initial direction is of increasing track number, then the first track selected is 150, since this is the closest track to the starting track of 100 in the increasing direction.

As can be seen, the SCAN policy behaves almost identically with the SSTF policy. Indeed, if we had assumed that the arm was moving in the direction of lower track numbers at the beginning of the example, then the scheduling pattern would have been identical for SSTF and SCAN. However, this is a static example in which no new items are added to the queue. Even when the queue is dynamically changing, SCAN will be similar to SSTF unless the request pattern is unusual.

Note that the SCAN policy is biased against the area most recently traversed. Thus it does not exploit locality as well as SSTF or even LIFO.

It is not difficult to see that the SCAN policy favors jobs whose requests are for tracks nearest to both innermost and outermost tracks and favors the latest-arriving jobs. The first problem can be avoided via the C-SCAN policy, while the second problem is addressed by the N-step-SCAN policy.

C-SCAN The C-SCAN (circular SCAN) policy restricts scanning to one direction only. Thus, when the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again. This reduces the maximum delay experienced by new requests. With SCAN, if the expected time for a scan from inner track to outer track is t, then the expected service interval for sectors at the periphery is $2t$. With C-SCAN, the interval is on the order of $t + s_{max}$, where s_{max} is the maximum seek time.

Figure 11.7d and Table 11.2d illustrate C-SCAN behavior. In this case the first three requested tracks encountered are 150, 160, and 184. Then the scan begins starting at the lowest track number, and the next requested track encountered is 18.

N-step-SCAN and FSCAN With SSTF, SCAN, and C-SCAN, it is possible that the arm may not move for a considerable period of time. For example, if one or a few processes have high access rates to one track, they can monopolize the entire device by repeated requests to that track. High-density multisurface disks are more likely to be affected by this characteristic than lower-density disks and/or disks with only one or two surfaces. To avoid this "arm stickiness," the disk request queue can be segmented, with one segment at a time being processed completely. Two examples of this approach are N-step-SCAN and FSCAN.

The N-step-SCAN policy segments the disk request queue into subqueues of length N. Subqueues are processed one at a time, using SCAN. While a queue is being processed, new requests must be added to some other queue. If fewer than N requests are available at the end of a scan, then all of them are processed with the next scan. With large values of N, the performance of N-step-SCAN approaches that of SCAN; with a value of $N = 1$, the FIFO policy is adopted.

FSCAN is a policy that uses two subqueues. When a scan begins, all of the requests are in one of the queues, with the other empty. During the scan, all new requests are put into the other queue. Thus, service of new requests is deferred until all of the old requests have been processed.

Animation
RAID

11.6 RAID

As discussed earlier, the rate in improvement in secondary storage performance has been considerably less than the rate for processors and main memory. This mismatch has made the disk storage system perhaps the main focus of concern in improving overall computer system performance.

As in other areas of computer performance, disk storage designers recognize that if one component can only be pushed so far, additional gains in performance are to be had by using multiple parallel components. In the case of disk storage, this leads to the development of arrays of disks that operate independently and in parallel. With multiple disks, separate I/O requests can be handled in parallel, as long as the data required reside on separate disks. Further, a single I/O request can be executed in parallel if the block of data to be accessed is distributed across multiple disks.

With the use of multiple disks, there is a wide variety of ways in which the data can be organized and in which redundancy can be added to improve reliability. This could make it difficult to develop database schemes that are usable on a number of platforms and operating systems. Fortunately, industry has agreed on a standardized scheme for multiple-disk database design, known as RAID (redundant array of independent disks). The RAID scheme consists of seven levels,[2] zero through six. These levels do not imply a hierarchical relationship but designate different design architectures that share three common characteristics:

1. RAID is a set of physical disk drives viewed by the operating system as a single logical drive.
2. Data are distributed across the physical drives of an array in a scheme known as striping, described subsequently.
3. Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure.

The details of the second and third characteristics differ for the different RAID levels. RAID 0 and RAID 1 do not support the third characteristic.

The term *RAID* was originally coined in a paper by a group of researchers at the University of California at Berkeley [PATT88].[3] The paper outlined various RAID configurations and applications and introduced the definitions of the RAID levels that are still used. The RAID strategy employs multiple disk drives and distributes data in such a way as to enable simultaneous access to data from multiple drives, thereby improving I/O performance and allowing easier incremental increases in capacity.

The unique contribution of the RAID proposal is to address effectively the need for redundancy. Although allowing multiple heads and actuators to operate simultaneously achieves higher I/O and transfer rates, the use of multiple devices increases the probability of failure. To compensate for this decreased reliability, RAID makes use of stored parity information that enables the recovery of data lost due to a disk failure.

We now examine each of the RAID levels. Table 11.4 provides a rough guide to the seven levels. In the table, I/O performance is shown both in terms of data transfer capacity, or ability to move data, and I/O request rate, or ability to satisfy

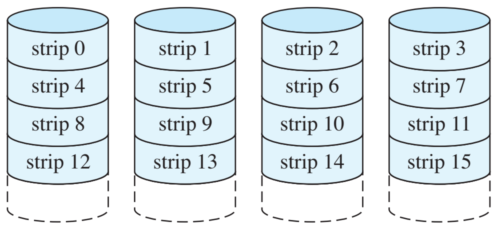
[2]Additional levels have been defined by some researchers and some companies, but the seven levels described in this section are the ones universally agreed on.

[3]In that paper, the acronym RAID stood for Redundant Array of Inexpensive Disks. The term *inexpensive* was used to contrast the small relatively inexpensive disks in the RAID array to the alternative, a single large expensive disk (SLED). The SLED is essentially a thing of the past, with similar disk technology being used for both RAID and non-RAID configurations. Accordingly, the industry has adopted the term *independent* to emphasize that the RAID array creates significant performance and reliability gains.

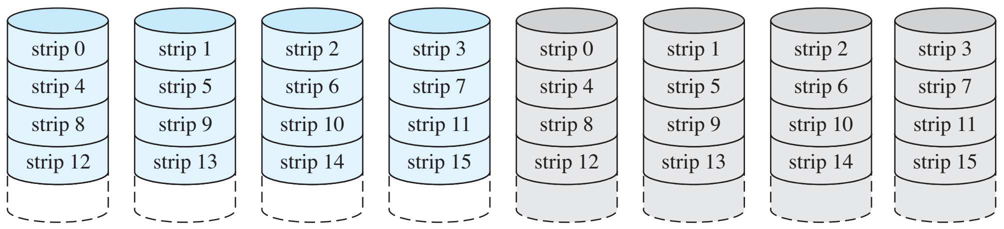
Table 11.4 RAID Levels

Category	Level	Description	Disks required	Data availability	Large I/O data transfer capacity	Small I/O request rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

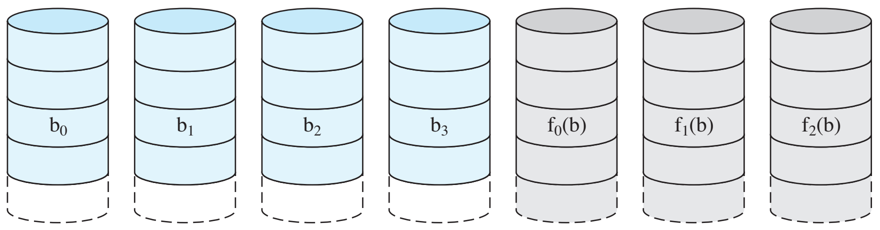
N = number of data disks; m proportional to log N



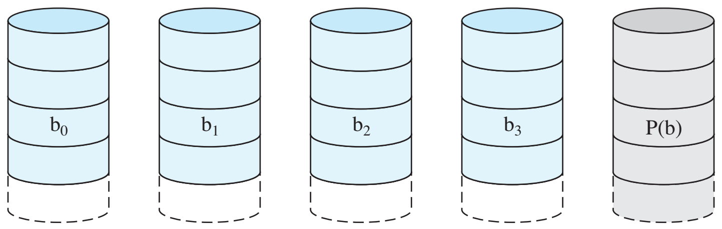
(a) RAID 0 (nonredundant)



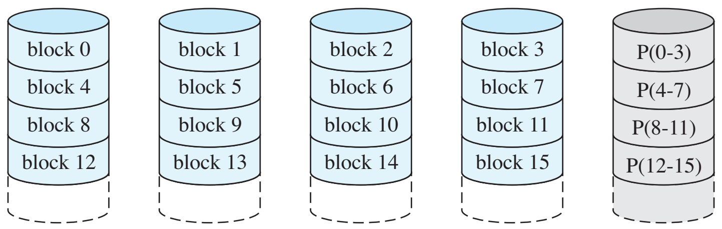
(b) RAID 1 (mirrored)



(c) RAID 2 (redundancy through Hamming code)

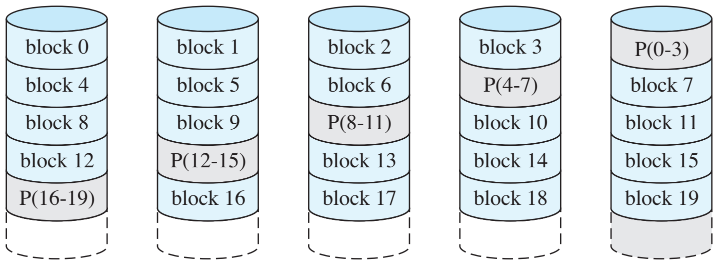


(d) RAID 3 (bit-interleaved parity)

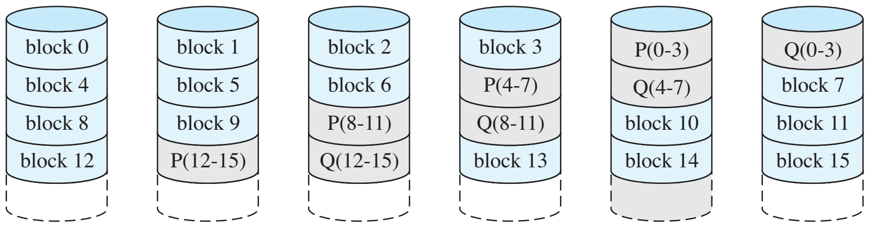


(e) RAID 4 (block-level parity)

Figure 11.8 RAID Levels



(f) RAID 5 (block-level distributed parity)



(g) RAID 6 (dual redundancy)

Figure 11.8 RAID Levels (continued)

I/O requests, since these RAID levels inherently perform differently relative to these two metrics. Each RAID level's strong point is highlighted in color. Figure 11.8 is an example that illustrates the use of the seven RAID schemes to support a data capacity requiring four disks with no redundancy. The figure highlights the layout of user data and redundant data and indicates the relative storage requirements of the various levels. We refer to this figure throughout the following discussion.

RAID Level 0

RAID level 0 is not a true member of the RAID family, because it does not include redundancy to improve performance. However, there are a few applications, such as some on supercomputers in which performance and capacity are primary concerns and low cost is more important than improved reliability.

For RAID 0, the user and system data are distributed across all of the disks in the array. This has a notable advantage over the use of a single large disk: If two different I/O requests are pending for two different blocks of data, then there is a good chance that the requested blocks are on different disks. Thus, the two requests can be issued in parallel, reducing the I/O queuing time.

But RAID 0, as with all of the RAID levels, goes further than simply distributing the data across a disk array: the data are *striped* across the available disks. This is best understood by considering Figure 11.8. All user and system data are viewed as being stored on a logical disk. The logical disk is divided into strips; these strips may be physical blocks, sectors, or some other unit. The strips are mapped round robin to consecutive physical disks in the RAID array. A set of logically consecutive strips that maps exactly one strip to each array member is referred to as a **stripe**.

In an *n*-disk array, the first *n* logical strips are physically stored as the first strip on each of the *n* disks, forming the first stripe; the second *n* strips are distributed as the second strips on each disk; and so on. The advantage of this layout is that if a single I/O request consists of multiple logically contiguous strips, then up to *n* strips for that request can be handled in parallel, greatly reducing the I/O transfer time.

RAID 0 for High Data Transfer Capacity The performance of any of the RAID levels depends critically on the request patterns of the host system and on the layout of the data. These issues can be most clearly addressed in RAID 0, where the impact of redundancy does not interfere with the analysis. First, let us consider the use of RAID 0 to achieve a high data transfer rate. For applications to experience a high transfer rate, two requirements must be met. First, a high transfer capacity must exist along the entire path between host memory and the individual disk drives. This includes internal controller buses, host system I/O buses, I/O adapters, and host memory buses.

The second requirement is that the application must make I/O requests that drive the disk array efficiently. This requirement is met if the typical request is for large amounts of logically contiguous data, compared to the size of a strip. In this case, a single I/O request involves the parallel transfer of data from multiple disks, increasing the effective transfer rate compared to a single-disk transfer.

RAID 0 for High I/O Request Rate In a transaction-oriented environment, the user is typically more concerned with response time than with transfer rate. For an individual I/O request for a small amount of data, the I/O time is dominated by the motion of the disk heads (seek time) and the movement of the disk (rotational latency).

In a transaction environment, there may be hundreds of I/O requests per second. A disk array can provide high I/O execution rates by balancing the I/O load across multiple disks. Effective load balancing is achieved only if there are typically multiple I/O requests outstanding. This, in turn, implies that there are multiple independent applications or a single transaction-oriented application that is capable of multiple asynchronous I/O requests. The performance will also be influenced by the strip size. If the strip size is relatively large, so that a single I/O request only involves a single disk access, then multiple waiting I/O requests can be handled in parallel, reducing the queuing time for each request.

RAID Level 1

RAID 1 differs from RAID levels 2 through 6 in the way in which redundancy is achieved. In these other RAID schemes, some form of parity calculation is used to introduce redundancy, whereas in RAID 1, redundancy is achieved by the simple expedient of duplicating all the data. Figure 11.8b shows data striping being used, as in RAID 0. But in this case, each logical strip is mapped to two separate physical disks so that every disk in the array has a mirror disk that contains the same data. RAID 1 can also be implemented without data striping, though this is less common.

There are a number of positive aspects to the RAID 1 organization:

1. A read request can be serviced by either of the two disks that contains the requested data, whichever one involves the minimum seek time plus rotational latency.
2. A write request requires that both corresponding strips be updated, but this can be done in parallel. Thus, the write performance is dictated by the slower of the

two writes (i.e., the one that involves the larger seek time plus rotational latency). However, there is no "write penalty" with RAID 1. RAID levels 2 through 6 involve the use of parity bits. Therefore, when a single strip is updated, the array management software must first compute and update the parity bits as well as updating the actual strip in question.

3. Recovery from a failure is simple. When a drive fails, the data may still be accessed from the second drive.

The principal disadvantage of RAID 1 is the cost; it requires twice the disk space of the logical disk that it supports. Because of that, a RAID 1 configuration is likely to be limited to drives that store system software and data and other highly critical files. In these cases, RAID 1 provides real-time backup of all data so that in the event of a disk failure, all of the critical data is still immediately available.

In a transaction-oriented environment, RAID 1 can achieve high I/O request rates if the bulk of the requests are reads. In this situation, the performance of RAID 1 can approach double of that of RAID 0. However, if a substantial fraction of the I/O requests are write requests, then there may be no significant performance gain over RAID 0. RAID 1 may also provide improved performance over RAID 0 for data transfer intensive applications with a high percentage of reads. Improvement occurs if the application can split each read request so that both disk members participate.

RAID Level 2

RAID levels 2 and 3 make use of a parallel access technique. In a parallel access array, all member disks participate in the execution of every I/O request. Typically, the spindles of the individual drives are synchronized so that each disk head is in the same position on each disk at any given time.

As in the other RAID schemes, data striping is used. In the case of RAID 2 and 3, the strips are very small, often as small as a single byte or word. With RAID 2, an error-correcting code is calculated across corresponding bits on each data disk, and the bits of the code are stored in the corresponding bit positions on multiple parity disks. Typically, a Hamming code is used, which is able to correct single-bit errors and detect double-bit errors.

Although RAID 2 requires fewer disks than RAID 1, it is still rather costly. The number of redundant disks is proportional to the log of the number of data disks. On a single read, all disks are simultaneously accessed. The requested data and the associated error-correcting code are delivered to the array controller. If there is a single-bit error, the controller can recognize and correct the error instantly, so that the read access time is not slowed. On a single write, all data disks and parity disks must be accessed for the write operation.

RAID 2 would only be an effective choice in an environment in which many disk errors occur. Given the high reliability of individual disks and disk drives, RAID 2 is overkill and is not implemented.

RAID Level 3

RAID 3 is organized in a similar fashion to RAID 2. The difference is that RAID 3 requires only a single redundant disk, no matter how large the disk array. RAID 3 employs parallel access, with data distributed in small strips. Instead of an error-correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks.

Redundancy In the event of a drive failure, the parity drive is accessed and data is reconstructed from the remaining devices. Once the failed drive is replaced, the missing data can be restored on the new drive and operation resumed.

Data reconstruction is simple. Consider an array of five drives in which X0 through X3 contain data and X4 is the parity disk. The parity for the *i*th bit is calculated as follows:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

where $\oplus$ is exclusive-OR function.

Suppose that drive X1 has failed. If we add $X4(i) \oplus X1(i)$ to both sides of the preceding equation, we get

$$X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

Thus, the contents of each strip of data on X1 can be regenerated from the contents of the corresponding strips on the remaining disks in the array. This principle is true for RAID levels 3 through 6.

In the event of a disk failure, all of the data are still available in what is referred to as reduced mode. In this mode, for reads, the missing data are regenerated on the fly using the exclusive-OR calculation. When data are written to a reduced RAID 3 array, consistency of the parity must be maintained for later regeneration. Return to full operation requires that the failed disk be replaced and the entire contents of the failed disk be regenerated on the new disk.

Performance Because data are striped in very small strips, RAID 3 can achieve very high data transfer rates. Any I/O request will involve the parallel transfer of data from all of the data disks. For large transfers, the performance improvement is especially noticeable. On the other hand, only one I/O request can be executed at a time. Thus, in a transaction-oriented environment, performance suffers.

RAID Level 4

RAID levels 4 through 6 make use of an independent access technique. In an independent access array, each member disk operates independently, so that separate I/O requests can be satisfied in parallel. Because of this, independent access arrays are more suitable for applications that require high I/O request rates and are relatively less suited for applications that require high data transfer rates.

As in the other RAID schemes, data striping is used. In the case of RAID 4 through 6, the strips are relatively large. With RAID 4, a bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk.

RAID 4 involves a write penalty when an I/O write request of small size is performed. Each time that a write occurs, the array management software must update not only the user data but also the corresponding parity bits. Consider an array of five drives in which X0 through X3 contain data and X4 is the parity disk. Suppose that a write is performed that only involves a strip on disk X1. Initially, for each bit i, we have the following relationship:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \tag{11.1}$$

After the update, with potentially altered bits indicated by a prime symbol:

$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \oplus X1(i) \oplus X1'(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

The preceding set of equations is derived as follows. The first line shows that a change in X1 will also affect the parity disk X4. In the second line, we add the terms $[\oplus\ X1(i) \oplus X1(i)]$. Because the XOR of any quantity with itself is 0, this does not affect the equation. However, it is a convenience that is used to create the third line, by reordering. Finally, Equation (11.1) is used to replace the first four terms by X4(i).

To calculate the new parity, the array management software must read the old user strip and the old parity strip. Then it can update these two strips with the new data and the newly calculated parity. Thus, each strip write involves two reads and two writes.

In the case of a larger size I/O write that involves strips on all disk drives, parity is easily computed by calculation using only the new data bits. Thus, the parity drive can be updated in parallel with the data drives and there are no extra reads or writes.

In any case, every write operation must involve the parity disk, which therefore can become a bottleneck.

RAID Level 5

RAID 5 is organized in a similar fashion to RAID 4. The difference is that RAID 5 distributes the parity strips across all disks. A typical allocation is a round-robin scheme, as illustrated in Figure 11.8f. For an n-disk array, the parity strip is on a different disk for the first n stripes, and the pattern then repeats.

The distribution of parity strips across all drives avoids the potential I/O bottleneck of the single parity disk found in RAID 4.

RAID Level 6

RAID 6 was introduced in a subsequent paper by the Berkeley researchers [KATZ89]. In the RAID 6 scheme, two different parity calculations are carried out and stored in separate blocks on different disks. Thus, a RAID 6 array whose user data require N disks consists of $N+2$ disks.

Figure 11.8g illustrates the scheme. P and Q are two different data check algorithms. One of the two is the exclusive-OR calculation used in RAID 4 and 5. But the other is an independent data check algorithm. This makes it possible to regenerate data even if two disks containing user data fail.

The advantage of RAID 6 is that it provides extremely high data availability. Three disks would have to fail within the MTTR (mean time to repair) interval to cause data to be lost. On the other hand, RAID 6 incurs a substantial write penalty, because each write affects two parity blocks. Performance benchmarks [EISC07] show a RAID 6 controller can suffer more than a 30% drop in overall write performance compared with a RAID 5 implementation. RAID 5 and RAID 6 read performance is comparable.

11.7 DISK CACHE

In Section 1.6 and Appendix 1A, we summarized the principles of cache memory. The term *cache memory* is usually used to apply to a memory that is smaller and faster than main memory and that is interposed between main memory and the processor. Such a cache memory reduces average memory access time by exploiting the principle of locality.

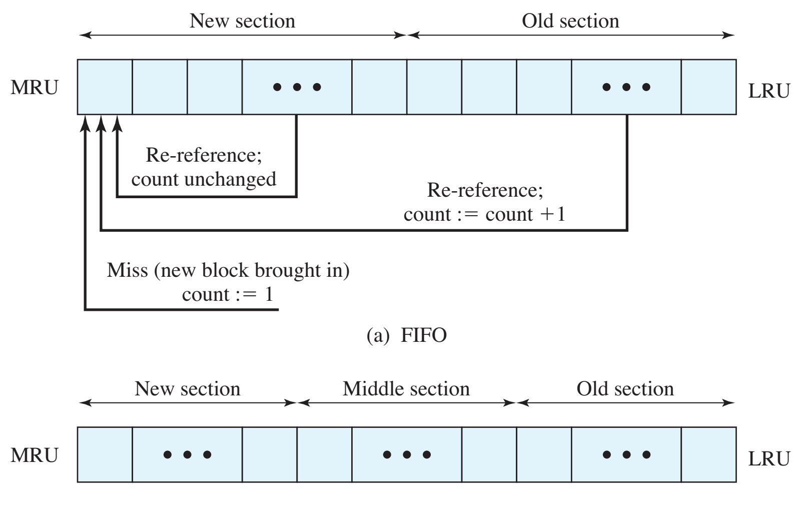
The same principle can be applied to disk memory. Specifically, a disk cache is a buffer in main memory for disk sectors. The cache contains a copy of some of the sectors on the disk. When an I/O request is made for a particular sector, a check is made to determine if the sector is in the disk cache. If so, the request is satisfied via the cache. If not, the requested sector is read into the disk cache from the disk. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single I/O request, it is likely that there will be future references to that same block.

Design Considerations

Several design issues are of interest. First, when an I/O request is satisfied from the disk cache, the data in the disk cache must be delivered to the requesting process. This can be done either by transferring the block of data within main memory from the disk cache to memory assigned to the user process, or simply by using a shared memory capability and passing a pointer to the appropriate slot in the disk cache. The latter approach saves the time of a memory-to-memory transfer and also allows shared access by other processes using the readers/writers model described in Chapter 5.

A second design issue has to do with the replacement strategy. When a new sector is brought into the disk cache, one of the existing blocks must be replaced. This is the identical problem presented in Chapter 8; there the requirement was for a page replacement algorithm. A number of algorithms have been tried. The most commonly used algorithm is least recently used (LRU): Replace that block that has been in the cache longest with no reference to it. Logically, the cache consists of a stack of blocks, with the most recently referenced block on the top of the stack. When a block in the cache is referenced, it is moved from its existing position on the stack to the top of the stack. When a block is brought in from secondary memory, remove the block that is on the bottom of the stack and push the incoming block onto the top of the stack. Naturally, it is not necessary actually to move these blocks around in main memory; a stack of pointers can be associated with the cache.

Another possibility is **least frequently used (LFU)**: Replace that block in the set that has experienced the fewest references. LFU could be implemented by associating a counter with each block. When a block is brought in, it is assigned a count of 1; with each reference to the block, its count is incremented by 1. When replacement is required, the block with the smallest count is selected. Intuitively, it might seem that LFU is more appropriate than LRU because LFU makes use of more pertinent information about each block in the selection process.



(b) Use of three sections

Figure 11.9 Frequency-Based Replacement

A simple LFU algorithm has the following problem. It may be that certain blocks are referenced relatively infrequently overall, but when they are referenced, there are short intervals of repeated references due to locality, thus building up high reference counts. After such an interval is over, the reference count may be misleading and not reflect the probability that the block will soon be referenced again. Thus, the effect of locality may actually cause the LFU algorithm to make poor replacement choices.

To overcome this difficulty with LFU, a technique known as frequency-based replacement is proposed in [ROBI90]. For clarity, let us first consider a simplified version, illustrated in Figure 11.9a. The blocks are logically organized in a stack, as with the LRU algorithm. A certain portion of the top part of the stack is designated the new section. When there is a cache hit, the referenced block is moved to the top of the stack. If the block was already in the new section, its reference count is not incremented; otherwise it is incremented by 1. Given a sufficiently large new section, this results in the reference counts for blocks that are repeatedly re-referenced within a short interval remaining unchanged. On a miss, the block with the smallest reference count that is not in the new section is chosen for replacement; the least recently used such block is chosen in the event of a tie.

The authors report that this strategy achieved only slight improvement over LRU. The problem is the following:

1. On a cache miss, a new block is brought into the new section, with a count of 1.
2. The count remains at 1 as long as the block remains in the new section.
3. Eventually the block ages out of the new section, with its count still at 1.
4. If the block is not now re-referenced fairly quickly, it is very likely to be replaced because it necessarily has the smallest reference count of those blocks that are not in the new section. In other words, there does not seem to be a sufficiently long interval for blocks aging out of the new section to build up their reference counts even if they were relatively frequently referenced.

A further refinement addresses this problem: divide the stack into three sections: new, middle, and old (Figure 11.9b). As before, reference counts are not incremented on blocks in the new section. However, only blocks in the old section are eligible for replacement. Assuming a sufficiently large middle section, this allows relatively frequently referenced blocks a chance to build up their reference counts before becoming eligible for replacement. Simulation studies by the authors indicate that this refined policy is significantly better than simple LRU or LFU.

Regardless of the particular replacement strategy, the replacement can take place on demand or preplanned. In the former case, a sector is replaced only when the slot is needed. In the latter case, a number of slots are released at a time. The reason for this latter approach is related to the need to write back sectors. If a sector is brought into the cache and only read, then when it is replaced, it is not necessary to write it back out to the disk. However, if the sector has been updated, then it is necessary to write it back out before replacing it. In this latter case, it makes sense to cluster the writing and to order the writing to minimize seek time.

Performance Considerations

The same performance considerations discussed in Appendix 1A apply here. The issue of cache performance reduces itself to a question of whether a given miss ratio can be achieved. This will depend on the locality behavior of the disk references, the replacement algorithm, and other design factors. Principally, however, the miss ratio is a function of the size of the disk cache. Figure 11.10 summarizes results from several studies using LRU, one for a UNIX system running on a VAX [OUST85]

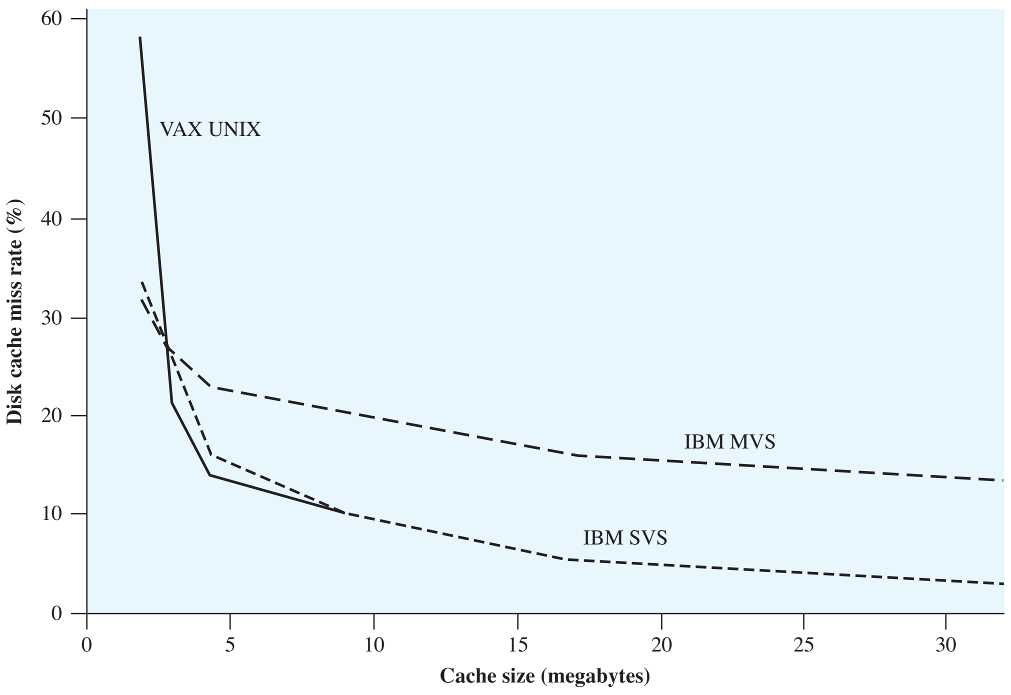


Figure 11.10 **Some Disk Cache Performance Results Using LRU**

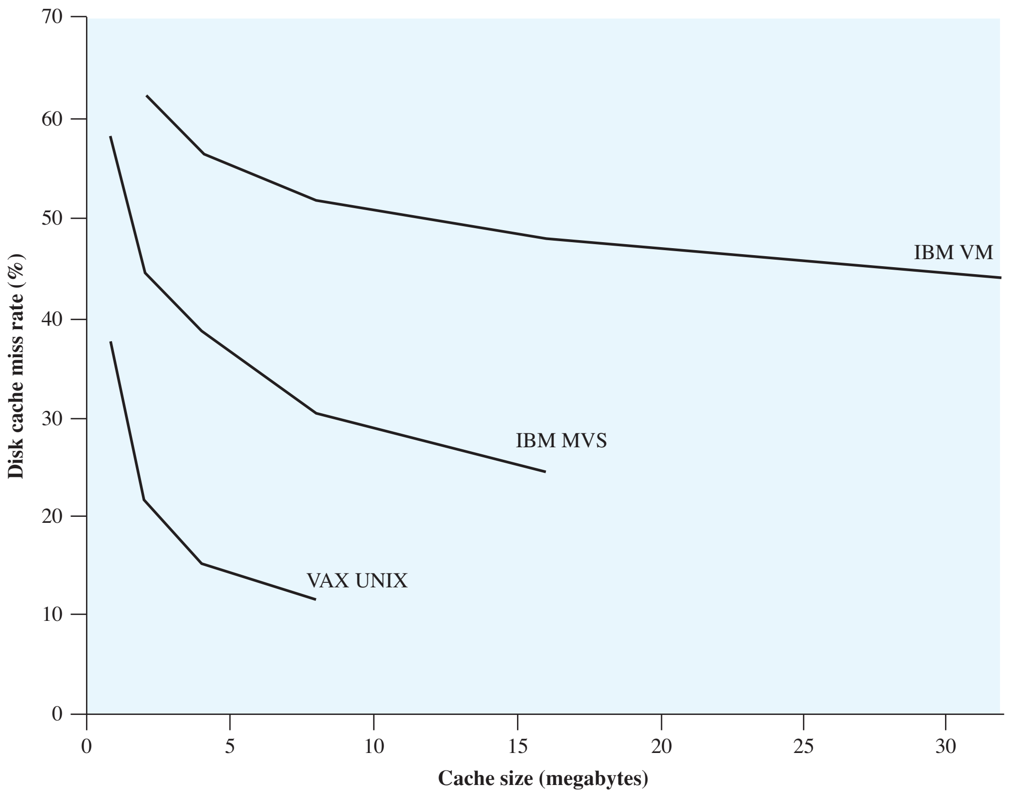


Figure 11.11 **Disk Cache Performance Using Frequency-Based Replacement**

and one for IBM mainframe operating systems [SMIT85]. Figure 11.11 shows results for simulation studies of the frequency-based replacement algorithm. A comparison of the two figures points out one of the risks of this sort of performance assessment. The figures appear to show that LRU outperforms the frequency-based replacement algorithm. However, when identical reference patterns using the same cache structure are compared, the frequency-based replacement algorithm is superior. Thus, the exact sequence of reference patterns, plus related design issues such as block size, will have a profound influence on the performance achieved.

11.13 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

block	disk pack	nonremovable disk
block-oriented device	fixed-head disk	programmed I/O
circular buffer	floppy disk	read/write head
CD-R	gap	redundant array of independent disks (RAID)
CD-ROM	hard disk	removable disk
CD-RW	interrupt-driven I/O	rotational delay
cylinder	input/output (I/O)	sector
device I/O	I/O buffer	seek time
digital versatile disk (DVD)	I/O channel	stream-oriented device
direct memory access (DMA)	I/O processor	track
disk access time	logical I/O	transfer time
disk cache	magnetic disk	
	movable-head disk	

Review Questions

11.1 List and briefly define three techniques for performing I/O.

11.2 What is the difference between logical I/O and device I/O?

11.3 What is the difference between block-oriented devices and stream-oriented devices? Give a few examples of each.

11.4 Why would you expect improved performance using a double buffer rather than a single buffer for I/O?

11.5 What delay elements are involved in a disk read or write?

11.6 Briefly define the disk scheduling policies illustrated in Figure 11.7.

11.7 Briefly define the seven RAID levels.

11.8 What is the typical disk sector size?

Problems

11.1 Consider a program that accesses a single I/O device and compare unbuffered I/O to the use of a buffer. Show that the use of the buffer can reduce the running time by at most a factor of two.

11.2 Generalize the result of Problem 11.1 to the case in which a program refers to n devices.

11.3 **a.** Perform the same type of analysis as that of Table 11.2 for the following sequence of disk track requests: 27, 129, 110, 186, 147, 41, 10, 64, 120. Assume that the disk head is initially positioned over track 100 and is moving in the direction of decreasing track number.

b. Do the same analysis, but now assume that the disk head is moving in the direction of increasing track number.

11.4 Consider a disk with N tracks numbered from 0 to $(N - 1)$ and assume that requested sectors are distributed randomly and evenly over the disk. We want to calculate the average number of tracks traversed by a seek.

a. First, calculate the probability of a seek of length j when the head is currently positioned over track t. (*Hint:* This is a matter of determining the total number of combinations, recognizing that all track positions for the destination of the seek are equally likely.)

b. Next, calculate the probability of a seek of length K. (*Hint:* This involves the summing over all possible combinations of movements of K tracks.)

c. Calculate the average number of tracks traversed by a seek, using the formula for expected value

$$\mathrm{E}[x] = \sum_{i=0}^{N-1} i \times \Pr[x = i]$$

Hint: Use the equalities $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$; $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.

d. Show that for large values of N, the average number of tracks traversed by a seek approaches $N/3$.

11.5 The following equation was suggested both for cache memory and disk cache memory:

$$T_S = T_C + M \times T_D$$

Generalize this equation to a memory hierarchy with N levels instead of just 2.

11.6 For the frequency based replacement algorithm (Figure 11.9), define F_{new}, F_{middle}, and F_{old} as the fraction of the cache that comprises the new, middle, and old sections, respectively. Clearly, $F_{new} + F_{middle} + F_{old} = 1$. Characterize the policy when

a. $F_{old} = 1 - F_{new}$

b. $F_{old} = 1/(\text{cache size})$

11.7 Calculate how much disk space (in sectors, tracks, and surfaces) will be required to store 300,000 120-byte logical records if the disk is fixed-sector with 512 bytes/sector, with 96 sectors/track, 110 tracks per surface, and 8 usable surfaces. Ignore any file header record(s) and track indexes, and assume that records cannot span two sectors.

11.8 Consider the disk system described in Problem 11.9, and assume that the disk rotates at 360 rpm. A processor reads one sector from the disk using interrupt-driven I/O, with one interrupt per byte. If it takes 2.5 μs to process each interrupt, what percentage of the time will the processor spend handling I/O (disregard seek time)?

11.9 Repeat the preceding problem using DMA, and assume one interrupt per sector.

11.10 A 32-bit computer has two selector channels and one multiplexor channel. Each selector channel supports two magnetic disk and two magnetic tape units. The multiplexor channel has two line printers, two card readers, and ten VDT terminals connected to it. Assume the following transfer rates:

Disk drive	800 Kbytes/s
Magnetic tape drive	200 Kbytes/s
Line printer	6.6 Kbytes/s
Card reader	1.2 Kbytes/s
VDT	1 Kbytes/s

Estimate the maximum aggregate I/O transfer rate in this system.

11.11 It should be clear that disk striping can improve the data transfer rate when the strip size is small compared to the I/O request size. It should also be clear that RAID 0 provides improved performance relative to a single large disk, because multiple I/O requests can be handled in parallel. However, in this latter case, is disk striping necessary? That is, does disk striping improve I/O request rate performance compared to a comparable disk array without striping?

11.12 Consider a 4-drive, 200GB-per-drive RAID array. What is the available data storage capacity for each of the RAID levels, 0, 1, 3, 4, 5, and 6?