CHAPTER 9

Storage Management

TABLE OF CONTENTS

The purpose of an operating system is to provide a convenient environment for higher-level programming and to allocate the resources of the system efficiently. Chapter 8 shows how the operating system allocates CPU time to the processes in the system. This chapter shows how it allocates space. The two primary classes of storage space are main memory and peripheral memory. Disk memory is the most common peripheral storage and is the type described here.

Two primary classes of storage space

9.1 Memory Allocation

Normally, programs that are not executing reside in a disk file. To execute, a program needs main memory space and CPU time. The operating system allocates space by loading the program from disk into main memory. It allocates time by setting the program counter to the address of the first instruction loaded into main memory.

The first two sections of this chapter describe five techniques for allocating main memory space:

Main memory allocation techniques

- Uniprogramming
- Fixed-partition multiprogramming
- Variable-partition multiprogramming
- Paging
- Virtual memory

The techniques are listed in order of increasing complexity. Each improves on the previous technique by solving a performance problem. A sixth technique, segmentation, is beyond the scope of this text.

Uniprogramming

The simplest memory allocation technique is *uniprogramming*, exemplified by the Pep/9 operating system. The operating system resides at one end of memory, and the application resides at the other end. The system executes only one job at a time.

Because every job will be loaded at the same place, the translators can generate object code accordingly. For example, when the Pep/9 assembler computes the symbols for the symbol table, it assumes that the first byte will be loaded at address 0000 (hex). Or, if the program contains a burn directive, it assumes that the last byte will be loaded at the address specified by the burn directive.

Advantages of uniprogramming

Uniprogramming has advantages and disadvantages. Its main advantage comes from its size. The system can be small, simple to design, and therefore relatively bug-free. It also executes with little overhead. Once an application is loaded, that application is guaranteed 100% of the processor's time, since no other process will interrupt it. Uniprocessing systems are appropriate for embedded systems like the ones that control a microwave oven.

Disadvantages of uniprogramming

The primary disadvantages are the inefficient use of CPU time and the inflexibility of job scheduling. Compared to main memory, disk memory has long access time. If the application executes a read from disk, the CPU will remain idle while waiting for the disk to deliver the input. The time could better be used executing another user's job. You can tolerate some waste of CPU time in a microcomputer, but in a computer that costs an organization hundreds of thousands of dollars you cannot, especially in a multiuser system in which other processes are executing concurrently.

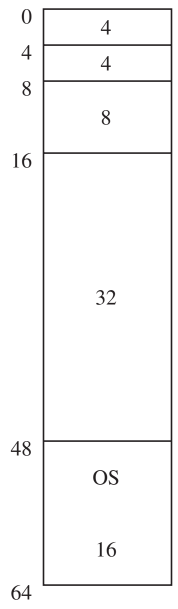
Even in a single-user system, the inflexibility of job scheduling can be a nuisance. The user may want to start up two programs and switch back and forth between them without quitting either one. For example, you may want to run a word processor for a while, switch to a drawing program to create an illustration for your document, and then switch back to the word processor where you left off to continue the text.

Fixed-Partition Multiprogramming

Multiprogramming solves the problem of inefficient CPU usage by allowing more than one application to run concurrently. To switch between two processes, the operating system loads both applications into main memory. When the running process is suspended, the operating system stores its process control block (PCB) and gives the CPU to the other process.

To implement multiprogramming, the operating system needs to partition main memory into different regions for storing the different processes while they execute. In a fixed-partition scheme, it subdivides memory into several regions whose sizes and locations do not change with time. FIGURE 9.1 shows one possible subdivision in a fixed-partition multiprogramming system with 64 KiB of main memory. The operating system occupies 16 KiB at the bottom of memory. It assumes that the jobs will not all be the same size, so it partitions the remaining 48 KiB into two 4-KiB regions, one 8-KiB region, and one 32-KiB region.

FIGURE 9.1 Fixed partitions in a 64-KiB main memory. Partition sizes and addresses are in kilobytes. The operating system occupies the bottom 16 KiB.



The address problem

A problem with providing different regions of memory for different processes is that the memory references in the object code must be adjusted accordingly. Suppose an assembly language programmer writes an application that is 20 KiB long, and the operating system loads it into the 32-KiB partition. If the assembler assumes that the object code will be loaded starting at address zero, all the memory references will be wrong.

For example, suppose the first few lines of code are

```
0000 040005         BR AbsVal
0003 0000   number: .BLOCK 2
0005 310003 AbsVal: DECI number,d
```

The assembler has computed the value of symbol `AbsVal` to be 0005 because `BR` is a three-byte instruction and `number` occupies two bytes. The problem is that `BR` branches to 0005, the address of the code for the process in the first partition. Not only would this process work incorrectly, but it also may destroy some data in the other process. The operating system needs to protect processes from unauthorized tampering by other concurrent processes.

Relocatable loaders

A loader that has the capability of loading a program anywhere in memory is called a *relocatable loader*. The Pep/9 loader described in Section 8.1 is not a relocatable loader because it loads every program into memory at the same location, namely 0000 (hex).

There are several approaches to this problem. The operating system could require the assembly language programmer to decide where in memory to load the application. The assembler would need a directive that allows the programmer to specify the address of the first byte of object code. A common designation for such a directive is `.ORG`, which means *origin*. In this example, the starting address of the 32-KiB partition is 16 Ki or 8000 (hex). The first few lines of the listing would be modified as follows:

```
                    .ORG   0x8000
8000 048005         BR     AbsVal
8003 0000   number: .BLOCK 2
8005 318003 AbsVal: DECI   number,d
```

The net effect is to add 8000 to all the memory references in the application code.

If a compiler generates the object code for an application, the programmer has no concept of memory addresses. The translator would need to cooperate with the operating system to generate the correct memory references in the object code.

Logical Addresses

Requiring the programmer or the compiler to specify in advance where the object code will be loaded has several drawbacks. An applications programmer should not have to worry about partition sizes and locations. That information is not relevant to the programmer. The scheme defeats the purpose of the operating system to provide a convenient environment for higher-level programming.

It also defeats the purpose of allocating the resources of the system efficiently. Suppose the programmer specifies a 3-KiB program to be loaded in the second 4-KiB partition at address 4 Ki and sets the `.ORG` directive accordingly. During the course of events, the job may be waiting to be loaded while another job occupies that partition, even though the first 4-KiB partition is free. The unused memory represents an inefficient allocation of a resource.

To alleviate these problems, the operating system can let the programmer or compiler generate the object code as if it will be loaded at address zero. An address generated under this assumption is called a *logical address*. If the program is loaded into a partition whose address is not zero, the operating system must translate logical addresses to *physical addresses*.

The following equation depicts the relationship between the physical address, logical address, and address of the first byte of the partition in which the program is loaded:

Logical address versus physical address

Physical address = logical address + partition address

An example is the previous code fragment, in which the logical address of `number` is 0003 and the physical address is 8003.

Two address-translation techniques are possible. The operating system could provide a software utility that adds the partition address to all the memory references in the object code. The translator would need to specify those parts of the object code that need to be adjusted, because the utility cannot tell by inspection of the raw object code which parts are memory references.

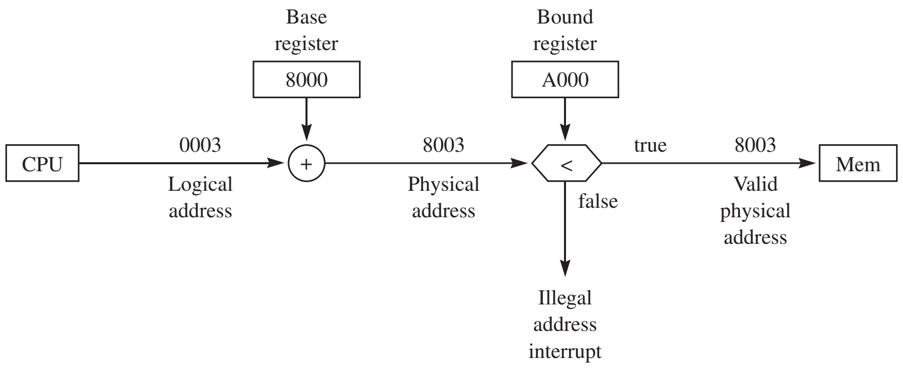
Operation of base and bound registers

Another technique depends on the availability of specialized hardware called *base* and *bound registers*. The base register solves the address-translation problem. The bound register solves the protection problem. **FIGURE 9.2** shows how base and bound registers work with the previous example.

The operating system loads the object program with unmodified logical addresses into the partition at 8000. It loads the base register with a value of 8000 and the bound register with a value of A000 (hex) = 48 Ki, the address of the upper bound of the partition in which the program is loaded. It turns the CPU over to the process by setting the program counter to 0000, the logical address of the first instruction.

Whenever the CPU issues a memory read request, the hardware adds the content of the base register to the address supplied by the CPU to form the physical address. It compares the physical address to the content of the bound register. If the physical address is less than the bound register, the hardware completes the memory access. Otherwise, it generates an illegal-address interrupt that the operating system must service. The bound register prevents a process from invading another process's memory partition.

FIGURE 9.2
Transformation of a logical address to a physical address with base and bound registers.



The first memory read request in this example will be from the fetch part of the von Neumann execution cycle. The CPU will request a fetch of the instruction from 0000, which the hardware will translate to a fetch from 8000. Figure 9.2 shows the translation of logical address 0003 from the `DECI` operand specifier. (Actually, at a lower level of abstraction, it is the operand of the `STWA` instruction at FEB7 in the `DECI` trap handler.)

To switch to another process, the operating system sets the base register to the address of the partition where the process has been loaded and the bound register to the address of the next-higher partition. When it restores the CPU registers (including the program counter) from the PCB, the process will continue executing from where it was suspended.

Scheduling problems with fixed partitions

Consider some of the problems that confront the operating system when it must schedule jobs to occupy the fixed partitions. Assume that all the partitions in Figure 9.1 are occupied except the 32-KiB partition. If a 4-KiB job requests execution, should the system put it in the 32-KiB partition or should it wait until a smaller partition becomes available? Suppose it starts the job in the 32-KiB partition, and then a 4-KiB process terminates. If a 32-KiB job now enters the system, it cannot be loaded. In retrospect, it would have been better to not schedule the small job in the large partition. Then the large job could use the large partition as soon as it requested execution, and the small job could be loaded soon anyway. Because the operating system cannot predict when a process will terminate or when a job will request execution, it cannot achieve the optimum schedule.

Another problem is how the operating system should set up the partitions in the first place. In Figure 9.1, if a 16-KiB job and a 32-KiB job request execution at the same time, only one can be loaded, even though a total of 48 KiB is available in user memory. On the other hand, if the operating system sets up user memory in two large partitions, and six or eight 4-KB jobs request execution, all but two will be delayed. Again, the optimum partition cannot be established because the operating system cannot predict the future.

Variable-Partition Multiprogramming

To alleviate the inefficiencies inherent in fixed-partition scheduling, the operating system can maintain partitions with variable boundaries. The idea is to establish a partition only when a job is loaded into memory. The size of the partition can exactly match the size of the job so more memory will be available to jobs as they enter the system.

When a job stops execution, the region of memory that it occupied becomes available for other jobs. A region of memory that is available for use by incoming jobs is called a *hole*. Holes are filled when the operating system allocates them to subsequent jobs. As in the fixed-partition scheme, the operating system attempts to schedule the jobs to maintain the largest number of processes in memory at any given point in time.

FIGURE 9.3 shows an example of a 48-KiB region of available memory before any jobs are scheduled. FIGURE 9.4 is a hypothetical sequence of job requests for the user memory in Figure 9.3. The table also indicates when a job stops executing, thereby releasing its memory for another job to use.

FIGURE 9.5 illustrates the scheduling process. The question that must be resolved is the selection criterion for determining which hole to fill when a new job requests some memory. Figure 9.5 uses what is called the *best-fit algorithm*. Of all the holes that are larger than the memory required for the job, the operating system selects the smallest. That is, the system selects the hole that the job fits best.

When J1 requests 12 KiB, there is only one hole from which to allocate memory, the initial hole in Figure 9.3. The system gives the first 12 KiB to J1, leaving a 36-KiB hole. When J2 requests 8 KiB, the system gives the first part of the smaller hole to it, and similarly for J3, J4, and J5. The result is the memory allocation in Figure 9.5(a).

Figure 9.5(b) shows the allocation when J1 stops executing, relinquishing its 12 KiB and creating a second hole at the top of main memory. When J6 requests a 4-KiB region, the operating system has the option of allocating memory for it from either hole. According to the best-fit algorithm, the system selects the 8-KiB hole rather than the 12-KiB hole, because the smaller hole is a better fit. Figure 9.5(c) shows the result.

Figure 9.5(d) shows the allocation after J5 stops, and (e) shows it after the system allocates memory from the top hole to J7. There are now three small holes scattered throughout memory. This phenomenon is called

FIGURE 9.3
The initial available user memory. Addresses are in kilobytes.

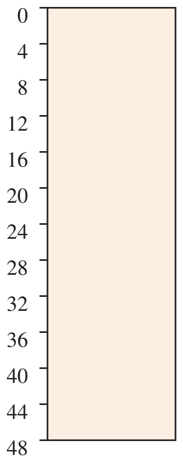
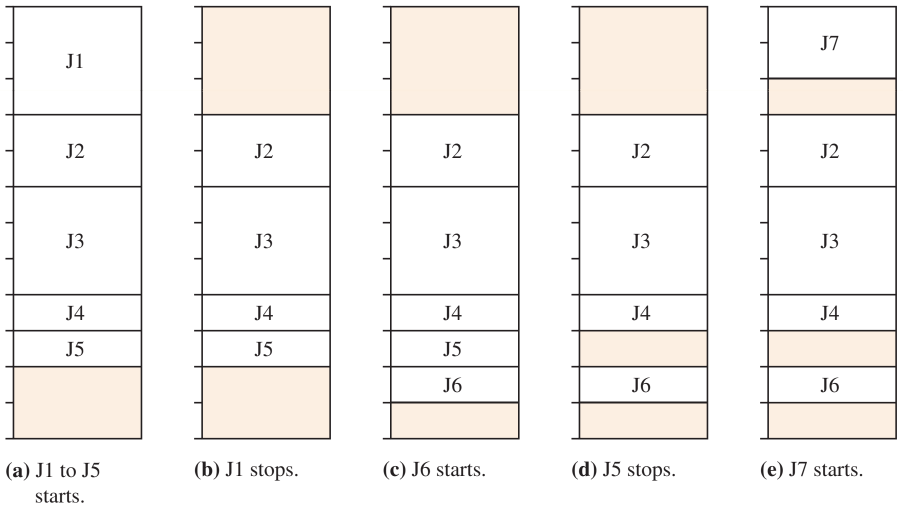


FIGURE 9.4
A job execution sequence for a variable-partition multiprogramming system. Job size is in kilobytes.

Job	Size	Action
J1	12	Start
J2	8	Start
J3	12	Start
J4	4	Start
J5	4	Start
J1	12	Stop
J6	4	Start
J5	4	Stop
J7	8	Start
J8	8	Start

FIGURE 9.5
The best-fit algorithm.



fragmentation. Even though J8 wants 8 KiB and available memory totals 12 KiB, J8 cannot run because the memory is not contiguous.

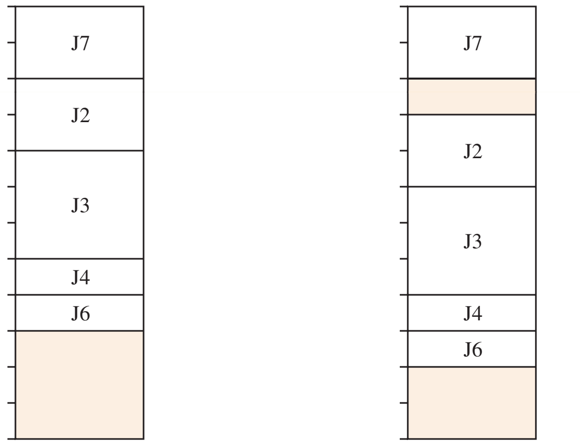
When confronted with a request that cannot be satisfied because of fragmented memory, the operating system could simply wait for enough processes to complete until a large enough hole becomes available. In this example, the request is so small that any job that completes will free enough memory for J8 to be loaded.

In a crowded system running many small jobs with a large request outstanding, the request may be pending for a long time before allocation. In such a case, the operating system may take time to move some processes to make a large enough hole to satisfy the request. This operation is called *compaction*.

FIGURE 9.6(a) shows the most straightforward compaction technique. The operating system shifts the processes up in memory, eliminating all the holes between them. Another possible compaction scheme is to move only enough processes to create a hole big enough to satisfy the request. In Figure 9.6(b), the system shifts only J6, leaving a large enough hole to load J8.

The idea behind the best-fit algorithm is to minimize fragmentation by using the smallest hole possible, leaving the larger holes available for future scheduling. Another scheduling technique that may not appear to be so reasonable is the *first-fit algorithm*. Rather than search for the smallest possible hole, this algorithm begins its search from the top of main memory, allocating

FIGURE 9.6
Compacting main memory.



(a) Shifting all jobs to the top.

(b) Shifting only J6.

memory from the first hole that can accommodate the request. FIGURE 9.7 is a trace of the first-fit algorithm for the request sequence of Figure 9.4.

Figures 9.7(a) and (b) are identical to the best-fit algorithm of Figure 9.5. In Figure 9.7(c), J6 requests a 4-KB partition. Rather than allocate from the smaller hole at the bottom of memory, the first-fit algorithm finds the hole at the top of memory first, from which it allocates storage for J6.

Figure 9.7(d) shows J5 terminating. In (e), the 8-KiB request from J7 is filled by the first available hole, which is between J6 and J2. When J8 requests 8 KiB, a hole is available, and the system does not need to compact memory.

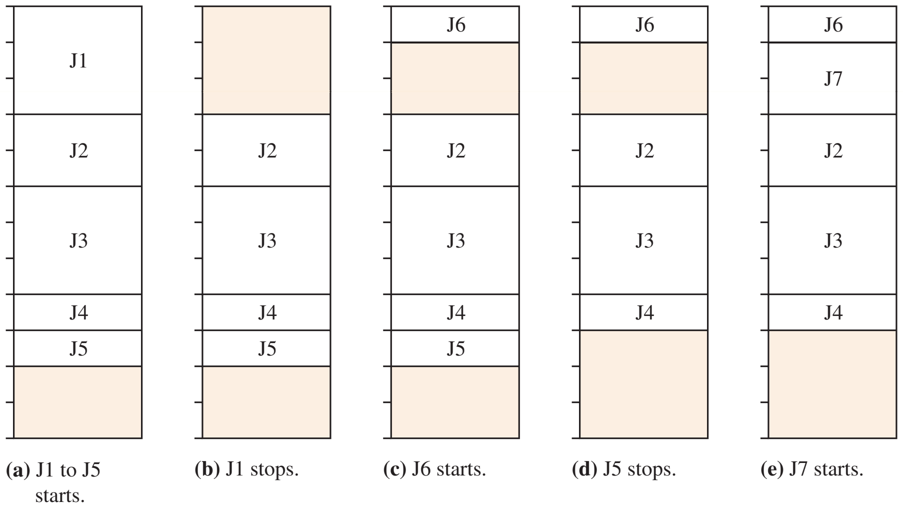
First fit versus best fit

One example does not prove that first fit is better than best fit. In fact, you can devise a sequence of requests and releases that will require compaction under the first-fit algorithm before best fit. The question is “What happens on the average?” It turns out that in practice, neither algorithm is substantially superior to the other in terms of memory utilization.

The reason first fit works so well is that storage tends to be allocated from the top of main memory. Therefore, large holes tend to form at the bottom of main memory. That is what happens in Figure 9.7.

Regardless of the allocation strategy, fragmentation is unavoidable in a variable-partition system. Noncontiguous holes represent an inefficient allocation of a resource. Even though unusable memory regions can be reclaimed with compaction, that is a time-consuming procedure.

FIGURE 9.7
The first-fit algorithm.



Paging

The idea behind paging

Paging is an ingenious idea to alleviate the fragmentation problem. Rather than coalesce several small holes to form one big hole for the program, paging fragments the program to fit the holes. Programs are no longer contiguous, but broken up and scattered throughout main memory.

FIGURE 9.8 shows three jobs executing in a paged system. Each job is subdivided into pages, and main memory is subdivided into frames that are the same size as the pages. The figure shows the first 12 KiB of a 64-KiB memory with 1-KiB frames. The page size is always a power of 2, in practice usually 4 KiB.

The code for job J3 is distributed to four noncontiguous frames in main memory. "J3, P0" in the frame at frame address 1800 represents page 0 of job J3. The second page is at 2C00, and the third and fourth pages are at 0800 and 2000, respectively. Jobs J1 and J2 are similarly scattered throughout memory. If job J4 comes along and needs 3 KB of memory, the operating system can distribute its pages to 0400, 1000, and 1400. The system does not need to compact memory to allocate it to the incoming job.

As with the previous multiprogramming memory management techniques, the application programs in paging assume logical addresses. The operating system must convert logical addresses to physical addresses during execution.

FIGURE 9.8
A paging system.

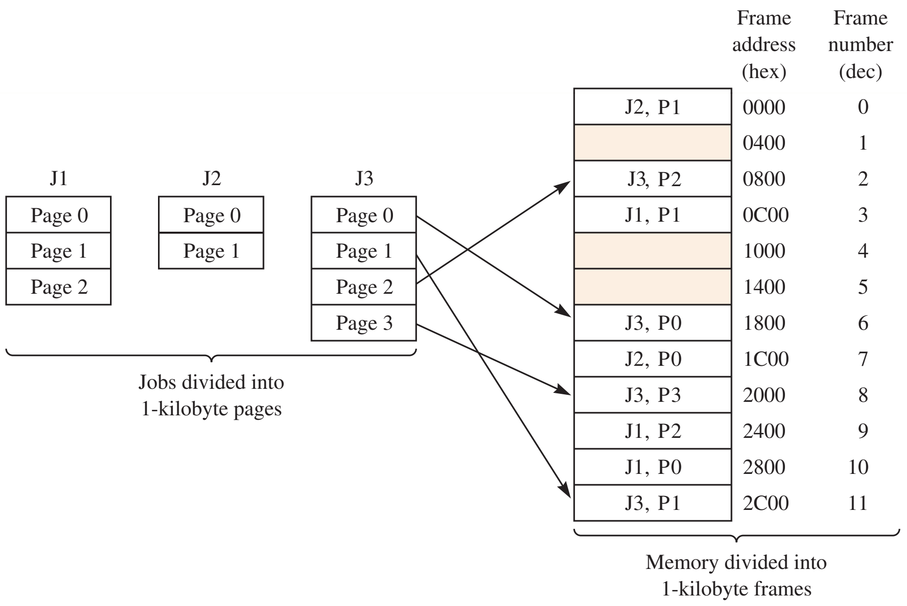


FIGURE 9.9
Logical and physical addresses in a paging system.

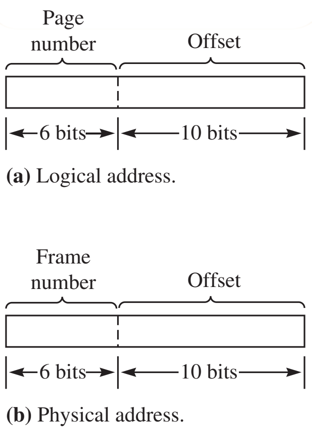


FIGURE 9.9 shows the relationship between a logical address and a physical address in the paged system of Figure 9.8. Because a page contains 1 KiB, which is 2^{10}, the rightmost 10 bits of a logical address are the offset from the top of the page. The leftmost 6 bits are the page number.

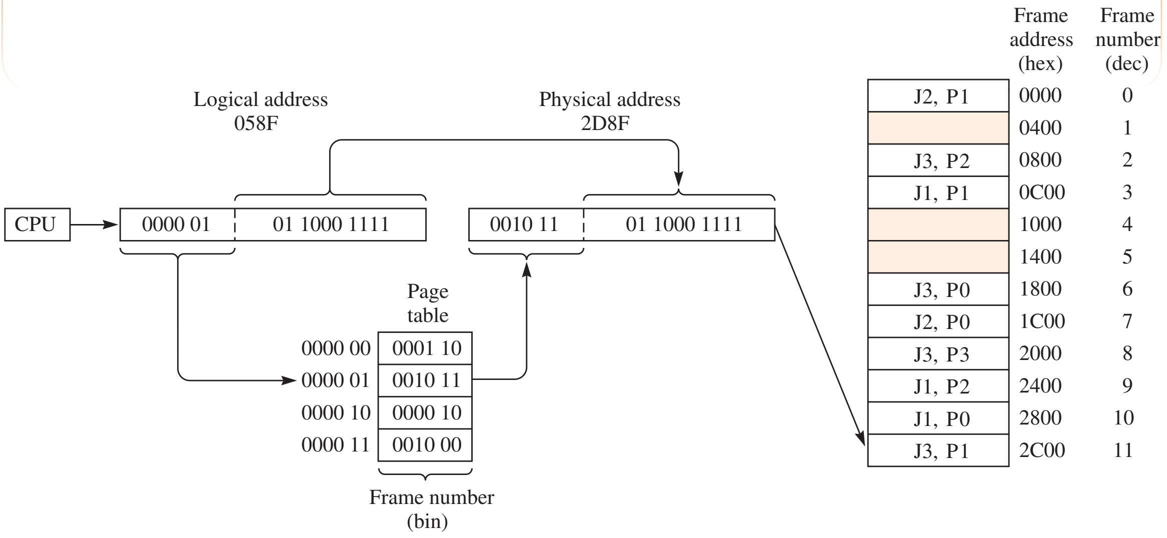
For example, consider the address 058F, which is 0000 0101 1000 1111 in binary. The leftmost six bits are 0000 01, which means that the address corresponds to a memory location in page number 1. Because 01 1000 1111 is 399 (dec), the logical address represents the 399th byte from the first byte in page number 1.

Referring to Figure 9.8, the physical address of this byte is the 399th byte from the first byte in the frame at address 2C00. To translate the logical address to the physical address, the operating system must replace the six-bit page number, 0000 01, with the six-bit frame number, 0010 11, leaving the offset unchanged.

Page tables

One base register was enough to transform a logical address to a physical address in the previous memory management schemes. Paging requires a set of frame numbers, however—one for each page of the job. Such a set is called a *page table*. FIGURE 9.10 shows the page table associated with job J3 of Figure 9.8. Each entry in the page table is the frame number that must replace the page number in the logical address.

FIGURE 9.10
Transformation of a logical address to a physical address with a page table.



Suppose job J3 executes the statement

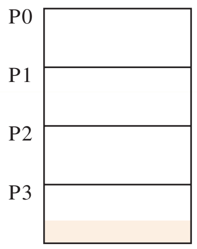
```
LDBA 0x058F,d
```

which causes the CPU to request a memory read from logical address 058F. The operating system extracts the first six bits, 0000 01, and uses them as an address in the page table, a special-purpose hardware memory that stores the frame numbers for the job. The frame number from the page table replaces the page number in the logical address to produce the physical address.

The byte is read from physical location 2D8F, even though the CPU issued a request for a read from location 058F. The program executes under the illusion that it has been loaded into a contiguous memory region starting at address 0000. The operating system perpetuates the illusion by maintaining a page table for each process loaded into memory. It is the ultimate scam. A process is continually interrupted in time and scattered through space without being aware of either indignity.

It is important to note that paging does not eliminate fragmentation altogether. It is rare that a job's size will be an exact multiple of the page size for the system. In that case, the last page will contain some unused memory, as FIGURE 9.11 shows for job J3. The unused memory at the end of the last page in a job is called *internal fragmentation*, in contrast to the external fragmentation that is visible to the operating system in the variable-partition scheme.

FIGURE 9.11
Internal fragmentation.



The smaller the page size, the less the internal fragmentation, on average. Unfortunately, there is a tradeoff. The smaller the page size, the greater the number of frames for a given main memory size and, therefore, the longer the page table. Because every reference to memory includes an access from a page table, the page tables usually are designed with the fastest possible circuitry. Such circuitry is expensive, so the page tables must be small to minimize the cost.

9.2 Virtual Memory

It may seem unlikely that you could improve on the memory utilization of a paged system, but the paging concept can be carried one step further. Consider the structure of a large program, say one that would fill 50 pages. To execute the program, is it really necessary for all 50 pages to be loaded into main memory at the same time?

Large Program Behavior

Most large programs consist of dozens of procedures, some of which may never execute. For example, procedures that are responsible for processing some input error condition will not execute if the input has no errors. Other procedures, such as those that initialize data, may execute only once and never be needed during the remainder of the execution.

A common control structure in any large program is the loop. As the body of a loop executes repeatedly, only that code in the loop need reside in main memory. Any code that is far from the loop (from an execution point of view) does not need to be in memory.

A program may also contain large regions of data that are never accessed. For example, if you declare an array of structures in C without knowing how many structures will be encountered when the program runs, you allocate more than you would reasonably expect to have. Pages that consist of unaccessed structures never need to be loaded.

These considerations of the typical large program show that it may be feasible to have only the active pages of the program loaded in memory. The active pages are those that contain code that is repeatedly executing and data that is repeatedly being accessed.

The working set

The set of active pages is called the *working set*. As the program progresses, new pages enter the working set and old ones leave. For example, at the beginning of execution, the pages that contain initialization procedures will be in the working set. Later, the working set will include the pages that contain the processing procedures and not the initialization procedures.

Virtual Memory

Remember that the programmer at a higher level of abstraction is under the illusion that the program executes in contiguous memory with logical addresses beginning at zero. Suppose the system can be designed to load only a few pages at a time from the executing job while still maintaining the illusion. It then becomes possible for the programmer to write a program that is too large to fit in main memory, but that will execute nonetheless. The user sees not the limited physically installed memory, but a virtual memory that is limited only by the virtual addresses and the capacity of the disk.

For example, in the older Pep/7 computer, an address contains 16 bits. It is, therefore, theoretically possible to access 2^{16} bytes (64 KiB) of memory. However, only 32 KiB of memory are installed. The application program starts at 0000 and cannot contain more than about 31,000 (dec) bytes without running into the operating system. It is common for a system to contain less memory than that permitted by the number of address bits. That situation allows the owner to upgrade the system at a later date by purchasing additional memory.

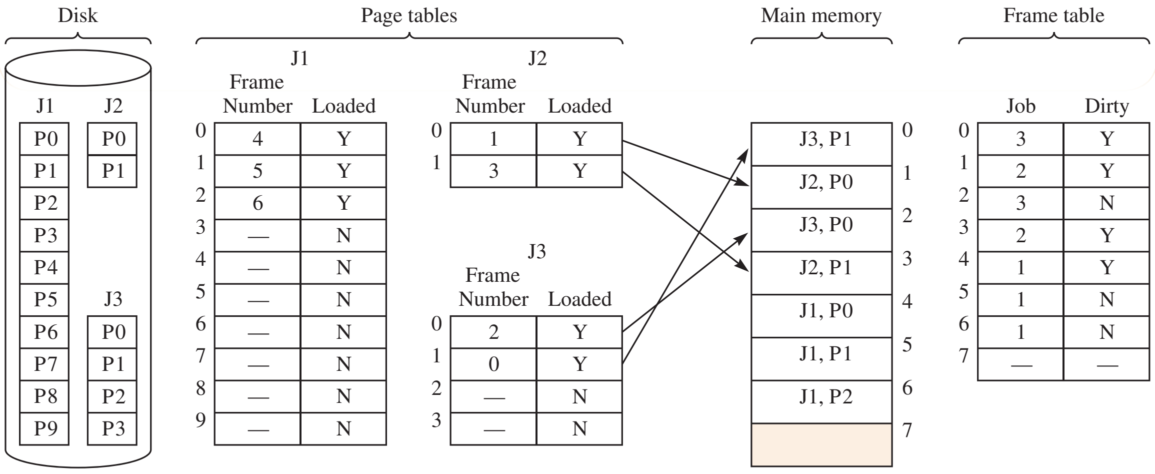
Suppose the Pep/7 computer has a virtual memory operating system. The program's physical memory is limited to 3000 bytes, but the programmer still could execute a 64-KiB program. The operating system loads the pages from disk into the memory frames as needed to execute the program. When a page needs to be loaded because it contains a statement to execute or some data to access, the operating system removes a page that is no longer active and replaces it with the one that needs to be loaded. The programmer sees the program execute in a 64-KiB virtual address space, even though the physical address space is 32 KiB.

FIGURE 9.12 shows how paging can be extended to implement a virtual memory system. It shows three jobs in the system, J1 with 10 pages, J2 with 2 pages, and J3 with 4 pages. Notice that physical memory contains only eight frames, but J1 can execute even though it is larger than physical memory. The special hardware required by the operating system includes a page table for each job and one frame table with an entry for each frame. To keep the illustration simple, frame numbers are given in decimal.

The page tables transform logical addresses to physical addresses, as in Figure 9.10. In a virtual memory system, however, some of the pages may not be loaded as the job is running. Each page table contains one extra bit per page that tells the operating system whether the page is loaded. The bit is 1 if the page is loaded and 0 if it is not. Figure 9.12 shows 1 as Y for yes, and 0 as N for no.

The frame table is needed to help the operating system allocate frames from main memory to the various jobs. The first entry is the job allocated to

FIGURE 9.12
An implementation of virtual memory.



	J1 Frame Number	J1 Loaded
0	4	Y
1	5	Y
2	6	Y
3	—	N
4	—	N
5	—	N
6	—	N
7	—	N
8	—	N
9	—	N

	J2 Frame Number	J2 Loaded
0	1	Y
1	3	Y

	J3 Frame Number	J3 Loaded
0	2	Y
1	0	Y
2	—	N
3	—	N

	Job	Dirty
0	3	Y
1	2	Y
2	3	N
3	2	Y
4	1	Y
5	1	N
6	1	N
7	—	—

that frame. The second entry is a bit called the *dirty bit*, whose function will be explained shortly.

Demand Paging

Figure 9.12 shows that jobs J1 and J3 do not have all their pages loaded into main memory. Suppose J3 is executing some code in page P1, loaded into frame 0. It will shortly execute an `LDWA` instruction whose operand is in P2. How will the operating system know that J3 is going to need P2 loaded into memory? Because the system cannot predict the future, it cannot know until J3 actually executes the `LDWA`.

In the course of translating the logical address to a physical address, the hardware accesses the page table for J3 to determine the frame number of the physical address. Because the loaded bit says *N*, an interrupt called a *page fault* occurs. The operating system intervenes to service the interrupt.

When a page fault occurs, the operating system searches the frame table to determine whether there are any empty frames in the system. Figure 9.12 shows that frame 7 is available, so the operating system can load P2 into that frame. It updates the frame table to show that frame 7 contains a page from J3. It updates the page table for J3 to show that P2 is in frame 7, and it sets the loaded bit to Y.

When the operating system returns from the interrupt, it sets the program counter to the address of the instruction that caused the page fault. That is, it restarts the instruction. This time, when the hardware accesses the page table for J3, an interrupt will not occur, and the operand of the LDWA instruction will be brought into the accumulator.

So when does the operating system load a page into main memory? The answer is, simply, when the program demands it. In the previous example, J3 demanded that P2 be loaded via the page fault interrupt mechanism. The difference between paging and demand paging is that demand paging brings pages into main memory only on demand. If a page is never demanded, it will never be loaded.

Page Replacement

When J3 demanded that P2 be loaded, the operating system had no problem because there was an empty frame in main memory. Suppose, however, that a job demands a page when all the frames are filled. In that case, the operating system must select a page that was previously loaded and replace it, freeing its frame for the demanded page.

The replaced page may subsequently be loaded again, perhaps into a different frame. To ensure that the page is reloaded in the same state that it was in when it was replaced, the operating system may need to save its state by writing the page to disk when the page is replaced. On the other hand, it may not be necessary to write the page to the disk when it is replaced.

Deciding whether the state of the replaced page needs to be updated on disk.

In Figure 9.12, J1 has 10 pages stored on disk, 3 of which have been loaded into main memory. When J1 executes instructions such as LDWA and ASLA, the effect is to not change the state of a page in main memory. LDWA issues a memory read and places the operand in the accumulator. ASLA changes the accumulator, an action that involves neither a memory read nor a memory write. Neither instruction issues a memory write.

But when J1 executes an instruction such as STWA, the instruction changes the state of the page in main memory. STWA puts the content of the accumulator in the operand, issuing a memory write in the process. If the operand is in P0 in frame 4, P0's state will change in main memory. Page P0 on disk will no longer be an exact copy of the current P0 in main memory. If no store instruction ever executes, the image of the page on disk will be an exact replica of the page in main memory.

When the operating system selects a page for replacement during a page fault, it does not need to write the page back to disk if the disk image is still a replica of the page in memory. To help the operating system decide whether the write is necessary, the hardware contains a special bit, called the *dirty bit*, in the frame table.

The dirty bit

When a page is first loaded into an empty frame, the operating system sets the dirty bit to 0, indicated by N in Figure 9.12. If a store instruction ever issues a write to memory, the hardware sets the dirty bit for that frame to 1, indicated by Y in the figure. Such a page is said to be dirty because it has been altered from its original clean state. If a page is selected for replacement, the operating system inspects the dirty bit to check whether it must write the page back to disk before overwriting the frame with the new page.

Page-Replacement Algorithms

The operating system has two memory management tasks in a demand paged system. It must allocate frames to jobs, and it must select a page for replacement when a page fault occurs and all the frames are full.

Frame allocation strategy

A reasonable allocation strategy for frames is to assume that a large job will need more frames than a small job. The system can allocate frames proportionally. If J1 is twice as big as J2, it gets twice as many frames in which to execute.

Page-replacement strategies

Given that a job has a fixed number of frames in which to execute, how does the operating system decide which page to replace when a page fault occurs and all the job's frames are full? Two possible page-replacement algorithms are *first-in, first-out* (FIFO) and *least recently used* (LRU).

FIGURE 9.13 shows the behavior of the FIFO page-replacement algorithm in a system that has allocated three frames to a job. As a job executes, the CPU sends a continuous stream of read and write requests to main memory. The first group of bits in each address is the page number, as Figures 9.9 and 9.10 show. The page references are the sequence of page numbers that the executing job generates.

Figure 9.13 shows three empty frames available before the first request. When the job demands P6, a page fault is generated, indicated by F in the figure, and P6 is loaded into a frame.

When the job demands P8, another page fault is generated and P8 is loaded into an empty frame. The boxes do not represent particular page frames. The figure shows P6 shifting to a lower box to accommodate P8. In the computer, P6 does not shift to another frame.

The reference to P3 causes another page fault, but the following reference to P8 does not because P8 is still in the set of loaded pages. Similarly, the reference to P6 does not produce a page fault.

FIGURE 9.13
The FIFO page-replacement algorithm with three frames.

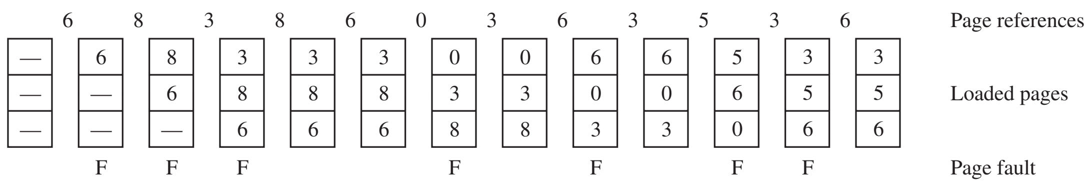
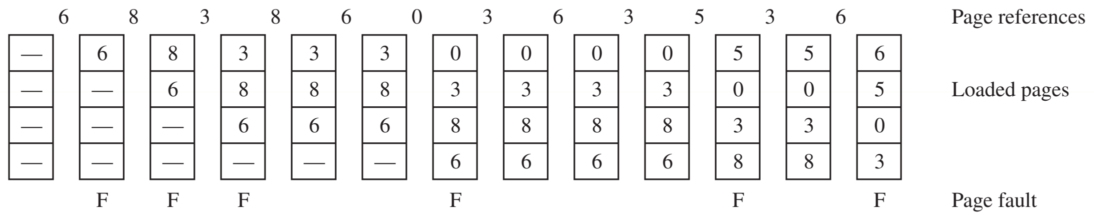


FIGURE 9.14
The FIFO page-replacement algorithm with four frames.



The reference to P0 triggers a page fault interrupt that must be serviced by selecting a replacement page. The FIFO algorithm replaces the page that was the first to enter the set. Because the figure shifts the pages down to accommodate a new page, the first page in is the one at the bottom, P6. The operating system replaces P6 with P0.

The given sequence of 12 page references produces 7 page faults when the job has three frames. If the job has more frames, the same sequence of page references should generate fewer page faults. FIGURE 9.14 shows the FIFO algorithm with the same page reference sequence but with four frames. As expected, the sequence generates fewer page faults.

In general, you would expect the number of page faults to decrease with an increase in the number of frames, as FIGURE 9.15 (a) shows, and as the two previous examples illustrate. Early in the development of demand paging systems, however, a curious phenomenon was discovered in which a given page reference sequence with the FIFO page-replacement algorithm actually produced more page faults with a greater number of frames.

A page reference sequence with this property is

0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4

Figure 9.15(b) is a plot of the number of page faults versus the number of frames for this sequence. It turns out that more page faults are generated with four frames than with three frames. This phenomenon is called *Bélády's anomaly* after L. A. Bélády, who discovered it.

Bélády's anomaly

The FIFO algorithm selects the page that has been in the set of frames the longest. That may appear to be a reasonable criterion. As the job executes, it will enter into new regions of code and data, so pages from the old region will no longer be needed. The oldest page is the one replaced.

On further reflection, however, it may be better to consider not how long a page has been in the set of frames, but how long it has been since a page was last referenced. The idea behind LRU is that a page referenced recently in the past is more likely to be referenced in the near future than a page that has not been referenced as recently.

FIGURE 9.16 illustrates the LRU page-replacement algorithm with the same sequence of page references as in Figure 9.13. The demands for P6, P8, and P3 produce a state identical to that of the FIFO algorithm. The next request for P8 brings that page to the top box to indicate that P8 is now the most-recently used. The following request for P6 brings it to the top, shifting down P8 and P3. The boxes maintain the pages in order of previous use, with the least recently used page at the bottom.

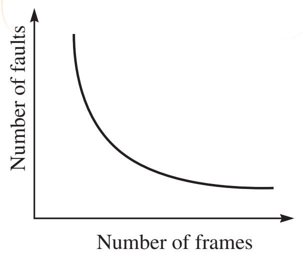
For this sequence, the LRU algorithm produced one fewer page fault than the FIFO algorithm did. One example does not prove that LRU is better than FIFO. It is possible to construct a sequence for which FIFO produces fewer faults than LRU.

In practice, operating systems have their own unique page-replacement algorithms that depend on the hardware features available on the particular computer. Most page-replacement algorithms are approximations to LRU, which generally works better than FIFO with the page request sequences from real jobs. An indication that LRU is better from a theoretical point of view is the fact that Bélády's anomaly cannot occur with LRU replacement.

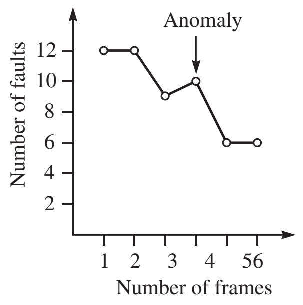
The sequences of page references in the previous examples only illustrate the page-replacement algorithms and are not realistic. For a demand paging system to be effective, the page fault rate needs to be kept to less than about one fault per 100,000 memory references.

A properly designed virtual memory system based on demand paging satisfies both goals of an operating system. It offers a convenient environment for higher-level programming because the programmer can develop code without being restricted by the limits of physical memory. It also allocates the memory efficiently because a job's pages are loaded only if needed.

FIGURE 9.15
The effect of more frames on the number of page faults.



(a) Expected effect of more frames on the number of page faults.

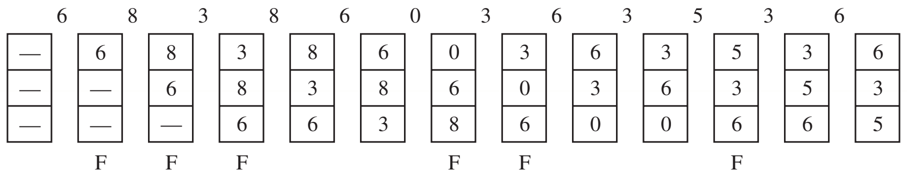


(b) Bélády's anomaly with the FIFO replacement algorithm.

9.3 File Management

The operating system is also responsible for maintaining the collection of files on disk. A file is an abstract data type (ADT). To the user of the system, a

FIGURE 9.16
The LRU page-replacement algorithm with three frames.



file contains a sequence of data and can be manipulated either by a program or by operating system commands. Common operations on files include

Common operations on files

- Create a new file.
- Delete a file.
- Rename a file.
- Open a file for reading.
- Read the next data item from the file.

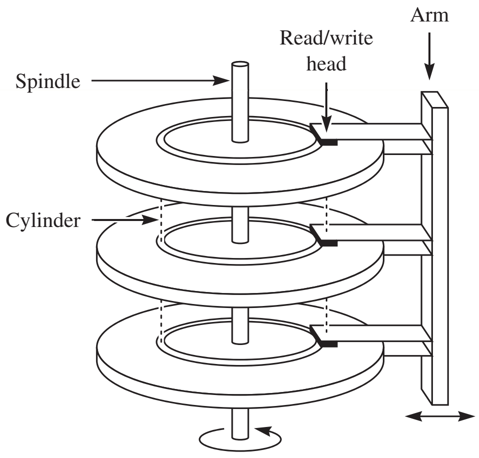
The operating system makes the connection between the logical organization of the file as seen by a programmer at Level HOL6 or Level Asmb5 and the physical organization on the disk itself.

Disk Drives

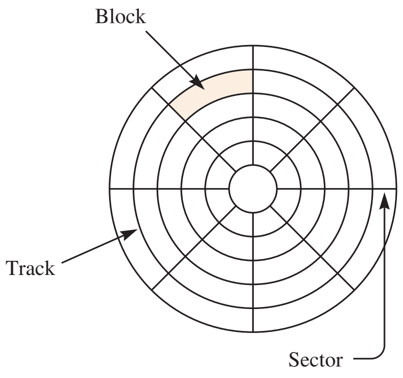
FIGURE 9.17 shows the physical characteristics of a disk drive. Figure 9.17(a) shows a hard disk drive that consists of several platters coated with magnetic recording material. They are attached to a central *spindle* that rotates at a typical speed of 7,200 revolutions per minute. Adjacent to each disk surface is a *read/write head* attached to an *arm*. The arm can move the heads in a radial direction across the surface of the platters.

Figure 9.17(b) shows a single disk. With the arm in a fixed position, the area under a read/write head sweeps out a ring as the disk rotates. Each

FIGURE 9.17
The physical characteristics of a disk drive.



(a) A hard disk drive.



(b) A single disk.

ring is a *track* that stores a sequence of bits. Tracks are divided into pie-shaped *sectors*. A *block* is one sector of one track of one surface. A *cylinder* is the set of tracks on all the surfaces at a fixed arm position. A *block address* consists of three numbers—a cylinder number, a surface number, and a sector number.

In a hard disk drive, the read/write heads float just above the surface on a small cushion of air. A *head crash* in a hard disk is a mechanical failure in which the head scrapes the surface, damaging the recording material.

Reading a block from a disk

Reading the information from a given block is a four-step process: (1) The arm must move the heads to the designated cylinder. (2) The electronic circuitry must select the read/write head on the designated surface. (3) A period of time must elapse for the designated block to reach the read/write head. (4) The entire block must pass beneath the head to be read. Step 2 is an electronic function, which occurs in negligible time compared to the other three steps.

Associated with the three mechanical steps are the following times:

Contributions to the disk access time

- Seek time
- Latency
- Transmission time

Seek time is the time it takes the arm to move to the designated cylinder. *Latency* is the time it takes the block to reach the head once the head is in place. *Transmission time* is the time it takes the block to pass beneath the head. The time it takes to access a block is the sum of these three times.

File Abstraction

The user at a high level of abstraction does not want to be bothered with physical tracks and sectors. The operating system hides the details of the physical organization and presents the file with a logical organization to the user as an ADT.

For example, in C, when you execute the statement

```
fscanf(fp, "%d", &myData)
```

where `fp` has type `FILE*`, you have a logical image of `fp` as a linear sequence of items with a current position maintained somewhere in the sequence. The `fscanf()` function gets the item at the current position and advances the current position to the next item in the sequence.

Physically, the items in the file may be on different tracks and surfaces. Furthermore, there is no physical current position that is maintained by the hardware. The logical behavior of the scan statement is due to the operating system software.

Allocation Techniques

The remainder of this section describes three memory allocation techniques at the physical level—contiguous, linked, and indexed. Each technique requires the operating system to maintain a directory that records the physical location of the files. The directory is itself stored on the disk, along with the files.

If each file were small enough to fit in a single block, the file system would be simple to maintain. The directory would simply contain a list of the files on the disk. Each entry in the directory would have the name of the file and the address of the block in which the file was stored.

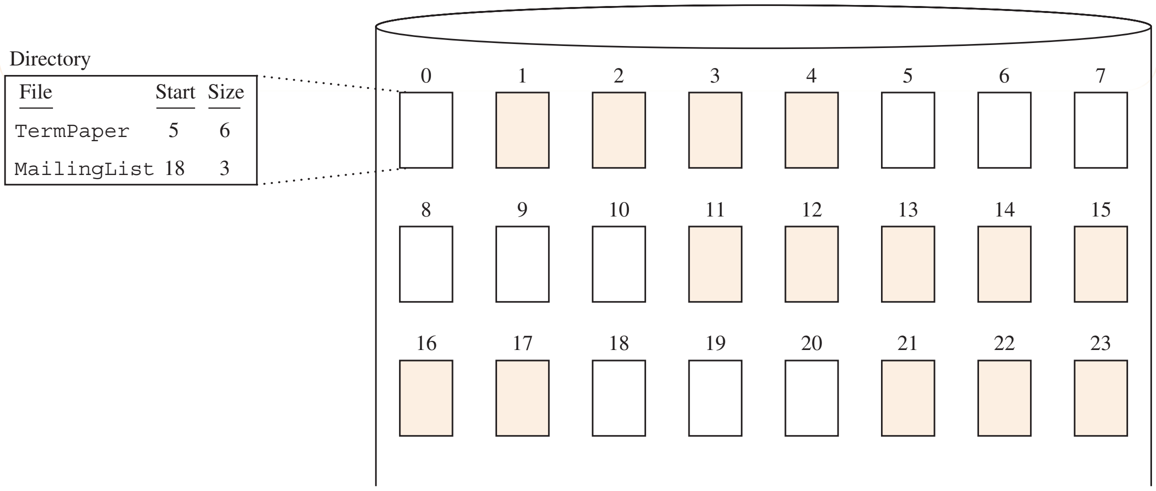
Contiguous allocation

If a file is too big to fit in a single block, the operating system must allocate several blocks for it. With *contiguous allocation*, the operating system matches the physical organization of the file to the logical organization by laying out the file sequentially on adjacent blocks of one track.

If the file is too big to fit on a single track, the system continues it on a second track. On a single-sided disk, the second track would be adjacent to the first track on the same surface. On a double-sided disk, the second track would be on the same cylinder as the first track. If the file is too big to fit on a single cylinder, the file would continue on an adjacent cylinder.

FIGURE 9.18 is a schematic diagram of contiguous allocation. Each row of eight blocks represents one track divided into eight sectors, as in Figure 9.17(b). The single number above each block is the block address, an

FIGURE 9.18
Contiguous allocation on a disk.



abbreviation for the three numbers required to specify the address. Block 0 contains the directory.

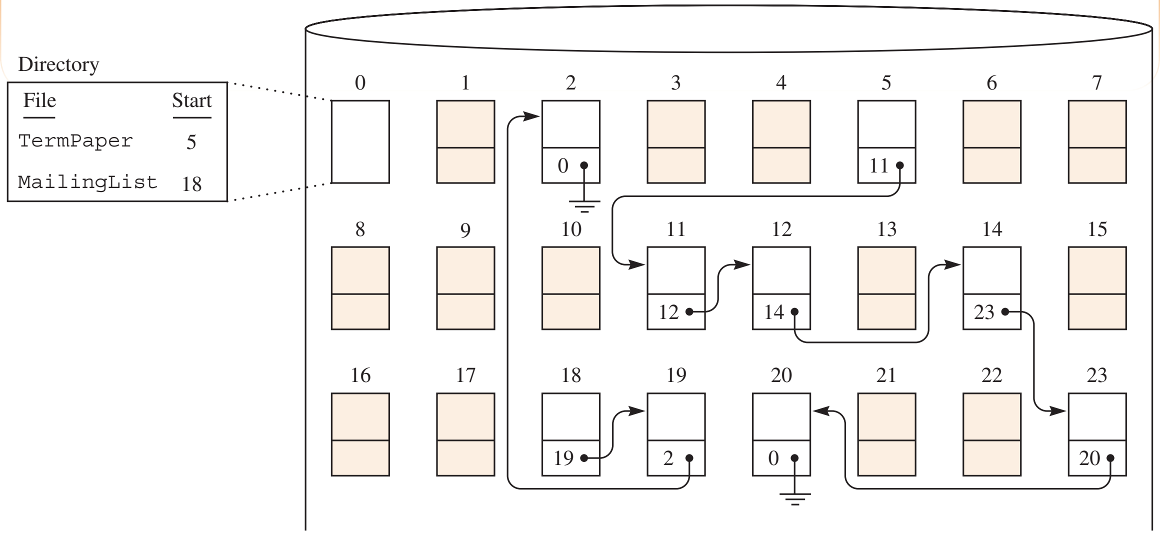
The directory lists the name of each file, its starting address, and its size. The file TermPaper starts at block 5 and contains six blocks. Its last three blocks are continued on a second track. Why wouldn't the system allocate blocks 1 through 6 for this file? The configuration in the figure could arise if another file previously occupied blocks 1 through 4, and then was deleted from the disk by the user.

The pattern of occupied and unoccupied disk memory in Figure 9.18 looks suspiciously like the pattern of occupied and unoccupied main memory in Figures 9.5 and 9.7. In fact, the memory management issues are the same. As files are created and deleted, they become fragmented. It may be impossible to create a new file because many small holes are scattered throughout the disk. To make room for the new file, the operating system supplies a disk compaction utility that shifts the files on the disk to make one large hole, as in Figure 9.6.

As with main memory, the compaction operation on disk is time-consuming. To eliminate the need for compaction, the operating system can store the file in blocks that are physically scattered throughout the disk. The linked allocation technique of FIGURE 9.19 is one way the system can maintain the file.

Linked allocation

FIGURE 9.19
Linked allocation on a disk.



The directory contains the address of the first block of the file. The last few bytes of each block are reserved for the address of the following block. The entire sequence of blocks forms a linked list. The link field in the last block has a nil value that acts like a sentinel.

One disadvantage of the linked technique is its susceptibility to failure. In Figure 9.19, suppose that just one byte in the link field of block 12 is damaged, either by a hardware failure or by a software bug. The operating system can still access the first three blocks of the file, but it will have no way of knowing where the last three blocks are.

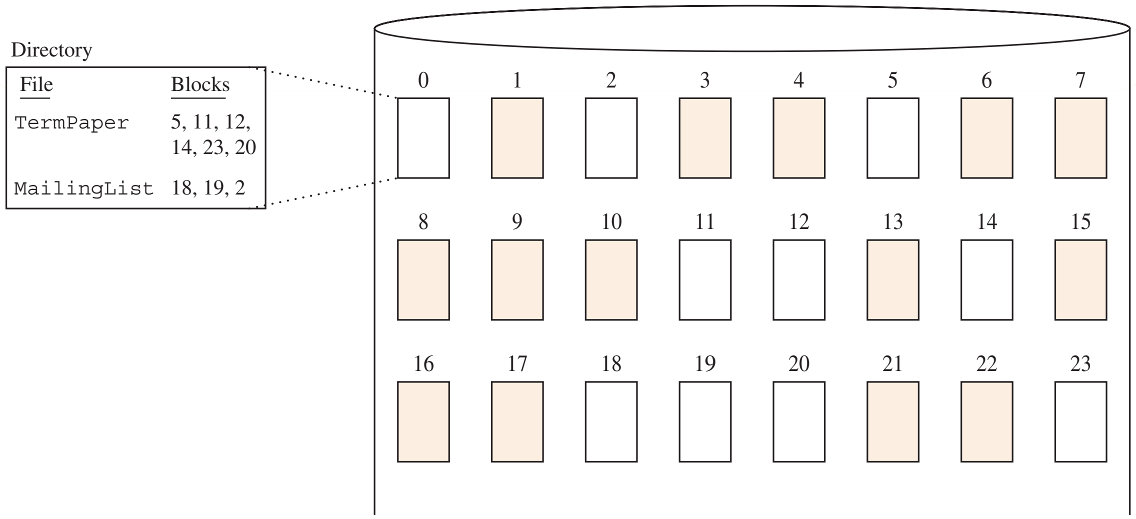
Indexed allocation

The indexed allocation technique in FIGURE 9.20 collects all the addresses into a single list called an *index* in the directory. Now, if a single byte in address 12 in the index is damaged, the operating system will lose track of only one block.

Contiguous allocation does have one major advantage over noncontiguous allocation: speed. If a file is contained on one cylinder, you only need to wait for one seek and one latency period to begin the access. You can read the entire file at a speed that is limited only by the transmission time. Even if the file is not all on one cylinder, after you read the first cylinder, you only need to wait for a short seek to an adjacent cylinder.

With the blocks of one file scattered throughout the disk, you must endure a seek time and a latency time to access each block. Even with noncontiguous allocation, it is sometimes worthwhile to periodically reorganize the physical layout of the files to make their blocks contiguous. This operation is called *defragmenting* the disk.

FIGURE 9.20
Indexed allocation on a disk.



9.4 Error-Detecting and Error-Correcting Codes

To be reliable, computer systems must be able to deal with the physical errors that inevitably happen in the real world. For example, if you send an email message over the Internet, there might be some static on the transmission lines that changes one or more of your bits. The result is that the receiver does not get the same bit pattern that you sent. As another example, the system might send some data from main memory to a disk drive, which due to a transient mechanical problem, might store an altered pattern on the disk.

There are two approaches to the error problem:

- Detect the error and retransmit or discard the received message.
- Correct the error.

Both approaches use the same technique of adding redundant bits to the message to detect or correct the error.

Error-Detecting Codes

Suppose you want to send a message about weather conditions. There are four possibilities—sunny, cloudy, raining, or snowing. The sender and receiver agree on the following bit patterns to encode the information:

00, sunny
01, cloudy
10, raining
11, snowing

It is raining, so the sender sends 10. But an error occurs on the transmission line that flips the last 0 to 1. So the receiver gets 11 and concludes erroneously that it is snowing.

A simple way to detect whether an error occurs is to append a redundant bit, called the *parity bit*, to the message using some computation that the sender and receiver agree upon. A common convention is to make the parity bit 0 or 1 in such a way that the total number of 1's is even. With this scheme, the sender and receiver agree on the following bit patterns, where the parity bit is underlined:

$00\underline{0}$, sunny
$01\underline{1}$, cloudy
$10\underline{1}$, raining
$11\underline{0}$, snowing

Now the sender would send 101 for the raining message. If an error flips the 0 to 1 so that the receiver gets 111, the receiver can conclude that an error occurred, because 111 is not one of the agreed-upon bit patterns. She can then request a retransmission or discard the received message.

Note that if the error occurs in the parity bit, the received message is just as useless as if it occurs in one of the data bits. For example, if the receiver gets 111, she does not know if the error was in the first bit with 011 sent, the second bit with 101 sent, or the third bit with 110 sent. She only knows that an error occurred.

The scheme would also work if the sender and receiver agreed to use odd parity, where the parity bit is computed to make the total number of 1's odd. The only necessity is for the sender and receiver to agree on the parity computation.

What if two errors occur during transmission so that not only is the 0 flipped to 1, but the last 1 is flipped to 0? Then the receiver gets 110. But now 110 is one of the agreed-upon patterns and the receiver concludes erroneously that it is snowing.

Codes and code words

The set of bit patterns {000, 011, 101, 110} is called a *code*, and an individual pattern from the set, such as 101, is called a *code word*. The above code cannot detect two errors. It is a single-error-detecting code. Error codes operate under the realistic assumption that the probability of error on a single bit is much less than 1.0. Hence, the probability of an error in two bits is much less than the probability of error in one bit. No code can completely eliminate the possibility of an undetectable error with 100% certainty, as it is always possible for multiple errors to occur that would change one code word into another code word. Error codes are still useful because they handle such a large percentage of error events.

Code Requirements

Suppose you want to be able to detect one or two errors. You will obviously need more parity bits. The questions are "How many parity bits?" and "How do you design the code?" The answers involve the concept of distance. The *Hamming distance* between two code words of the same length is defined as the number of positions in which the bits differ. It is named after Richard Hamming, who developed the theory in 1950 at Bell Labs.

The Hamming distance

Example 9.1 The Hamming distance between the code words for *cloudy*, 011, and *raining*, 101, is 2, because the code words differ in two positions—namely, the first and second positions. ■

Inspection of the weather code {000, 011, 101, 110} should convince you that the distance between all possible pairs of code words is also 2. You can see now that a code to detect a single error cannot have any pair of code words that are separated by a distance of 1. Suppose there are two such code words, A and B. Then it would be possible for the sender to send A, have an error in transmission that flipped the single bit where A and B differ, and have the receiver conclude that B was sent. The code would fail to detect the single error.

The code distance

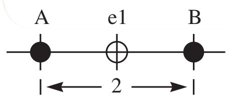
The *code distance* is the minimum of the Hamming distance between all possible pairs of code words in the code.

Example 9.2 The code {00110, 11100, 01010, 11101} has a code distance of 1. Although several code words, such as 00110 and 11101, are separated by a Hamming distance as great as 4, there exists a pair of words that are separated by a distance of only 1—namely, 11100 and 11101. If you used this code for sending the weather information, you could not guarantee the detection of all possible single-transmission errors. ▮

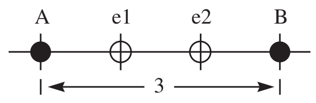
To design a good code, you must add parity bits in such a way that you make the code distance as large as possible. To detect one error, the code distance must be 2. What must the code distance be to detect two errors? The code distance cannot be 2, as that would mean that there exists a pair of code words A and B with a Hamming distance of 2. The sender could send A, have an error in transmission that flipped both bits where it differs from B, and have the receiver conclude that B was sent.

FIGURE 9.21 is a schematic representation of this concept. A is the transmitted code word and B is the code word closest to A. The open circles in between represent words not sent but possibly received because of errors in transmission. In Figure 9.21(a), e1 comes from a single error. In Figure 9.21(b), e1 comes from a single error and e2 comes from a double error.

FIGURE 9.21 Minimum Hamming distances for error-detecting codes.



(a) Single-error detecting.



(b) Double-error detecting.

The requirement to detect d errors

In general, to detect *d* errors, the code distance must satisfy the equation

$$\text{code distance} = d + 1$$

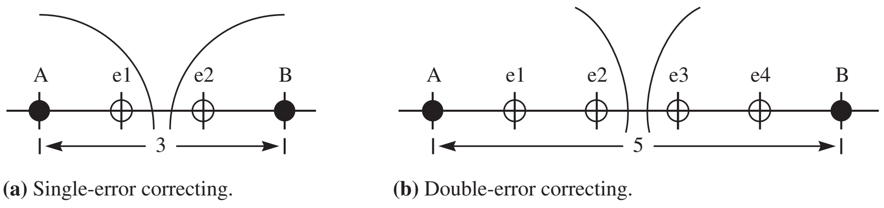
For example, to be able to detect three errors, the code distance must be 4. The reason is that with a distance of at least $d + 1$ to the closest code word to A, it is impossible to transform A to any other code word by flipping *d* bits of A.

The concept of distance is also useful with error-correcting codes. Suppose you decide on the following code for the weather messages:

00000, sunny
01101, cloudy
10110, raining
11011, snowing

If you receive 11110, what do you conclude? You could conclude that 00000 was sent for *sunny* and that four errors occurred. But is that a reasonable

FIGURE 9.22
Minimum Hamming distances for error-correcting codes.



conclusion? Not really, because 11110 is closer to 10110, the code word for *raining*. With that conclusion, you assume that only one error occurred, an event of much higher probability than the event of four errors.

In general, for an error-correction code, you add enough parity bits to make the code distance large enough that the receiver can correct the errors. The receiver corrects the errors by computing the Hamming distance between the received word and every code word, and picking the code word that is closest to the received word. "Close" is defined in terms of Hamming distance.

FIGURE 9.22 is a schematic representation of the error-correction concept. As before, A is the transmitted code word, B is the code word closest to A, and the open circles are words received because of errors in transmission. Figure 9.22(a) shows the situation for a code that is capable of correcting a single error. The code distance is 3, so that even if a single error occurs, the received word e1 will be closer to A than to B, and the receiver can conclude that A was sent. If A is sent and two errors occur, so that e2 is received, then the receiver will erroneously conclude that B was sent.

Figure 9.22(b) shows a code capable of correcting two errors. If A is sent and two errors occur, so that e2 is received, e2 is still closer to A than to B. That can happen only if the distance is 5.

In general, to correct d errors, the code distance must satisfy the equation

The requirement to correct d errors

$$\text{code distance} = 2d + 1$$

For example, to be able to correct three errors, the code distance must be 7. The reason is based on the decision process. The receiver concludes that A was sent when it receives words close to A. But it concludes that B was sent when it receives words close to B. The line between A and B must accommodate *both* sets of received words; hence the factor of 2 in the equation. Also, the distance must be odd; hence the +1 in the equation. If the distance were even, there would be a received word equidistant between A and B, and the receiver could not conclude which code word had the higher probability of being sent.

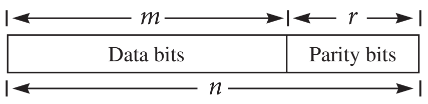
A code that can correct single errors can alternatively be used to detect double errors, as both have a code distance of 3. It is simply a question of how the receiver wants to handle the error condition. It can correct the error assuming two errors did not occur, or it can be more conservative, assume that two errors might have occurred, and discard the message or request a retransmission.

Single-Error-Correcting Codes

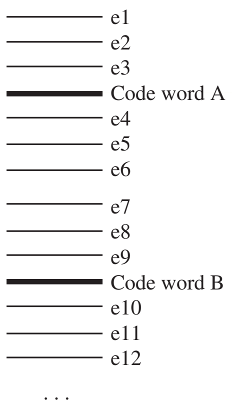
The previous section describes the requirements on the code distance for error detecting and correcting codes. The question remains of how to pick the code words to achieve the required code distance. Many different schemes have been devised for codes that correct multiple errors. This section investigates the efficiency of single-error-correcting codes and describes one systematic way to construct them.

FIGURE 9.23(a) shows the structure of a code word. There are m data bits and r parity bits, for a total of $n = m + r$ bits in the code word. Because there are n bits in a code word, there are 2^n possible received patterns. Figure 9.23(b) shows a schematic of how you can group those words that have no errors or one error. The figure shows the pattern for $n = 6$, where e1, e2, e3, e4, e5, and e6 are the six possible received words that could differ from A by one bit. If one of these is received, the receiver concludes that A was sent. Similarly, e7, e8, e9, e10, e11, and e12 are those possible words with a distance of 1 from code word B. There might be other received words

FIGURE 9.23
The single-error-correcting code structure.



(a) Code word structure.



(b) Grouping of received words with zero or one error.

that are not included in the grouping, corresponding to the event of more than one error during transmission, but resulting in a received word that is not within a distance of 1 from any code word.

In general, there are n words a distance of 1 from A. So the total number of words, including A, in the first group is $(n + 1)$. Similarly, there are $(n + 1)$ words in the B group, the C group, and so on. There is one group for each code word. So, as there are 2^m code words, there are 2^m groups. The total number of words in Figure 9.23(b) is, therefore, $(n + 1)2^m$. There could be other received words not in Figure 9.23(b), but there cannot be more than 2^n words altogether. Therefore,

$$(n + 1)2^m \le 2^n$$

Substituting $n = m + r$ and dividing both sides by 2^m gives

$$m + r + 1 \le 2^r$$

which tells how many parity bits r are necessary to correct a single error in a message with m data bits.

Perfect codes

A code for which the relationship holds with equality is called a *perfect code*. An example of a perfect code is $m = 4$, $r = 3$. Sending parity bits along with data bits increases the transmission time. For this code, for every four data bits, you must send an additional three parity bits. So, the error correction has added 3/4 = 75% overhead to the transmission time. If you need to send a long stream of bits, you must subdivide the stream into chunks and apply the parity bits to each chunk. The bigger the chunk, the smaller the overhead. With computers, you usually send streams of bytes, so the chunks are usually powers of 2. FIGURE 9.24 shows the relationship between m and r for a few values of m that are powers of 2.

FIGURE 9.24
The cost of a single-error-correcting code.

Data Bits m	Parity Bits r	Percent Overhead
4	3	75
8	4	50
16	5	31
32	6	19
64	7	11
128	8	6

FIGURE 9.25
The position of the four parity bits in a single-error-correcting code with eight data bits.

1	2	3	4	5	6	7	8	9	10	11	12
▨	▨	1	▨	0	0	1	▨	1	1	0	0

Hamming devised an ingenious technique for determining the parity bits of a single-error-correcting code. The idea is to not append the parity bits to the end of the code word, but to distribute them throughout the code word. The advantage of this technique is that the receiver can calculate directly which bit is the erroneous one without having to compute the distance between the received word and all the code words. FIGURE 9.25 shows the positions of the parity bits for the $m = 8$, $r = 4$ case. The bit positions are numbered consecutively from the left, and the parity bits are at locations 1, 2, 4, and 8, all powers of 2. In this example, the data to be transmitted is 1001 1100, but these bits are not stored contiguously in the code word.

The numeric position of each bit can be written as a unique sum of powers of 2 as follows:

$1 = 1$	$5 = 1 + 4$	$9 = 1 + 8$
$2 = 2$	$6 = 2 + 4$	$10 = 2 + 8$
$3 = 1 + 2$	$7 = 1 + 2 + 4$	$11 = 1 + 2 + 8$
$4 = 4$	$8 = 8$	$12 = 4 + 8$

To determine the parity bit at position 1, note that 1 occurs in the sum on the right-hand side for positions 1, 3, 5, 7, 9, and 11. Using even parity, set the parity bit so that the total number of 1's in those positions is even. There are 1's at positions 3, 7, and 9, an odd number of 1's. So, make the parity bit at position 1 a 1. The positions checked by each parity bit are:

Parity bit 1 checks 1, 3, 5, 7, 9, 11
Parity bit 2 checks 2, 3, 6, 7, 10, 11
Parity bit 4 checks 4, 5, 6, 7, 12
Parity bit 8 checks 8, 9, 10, 11, 12

You should verify that a similar computation for the other parity bits results in the code word $\underline{1}\ \underline{1}\ 1\ \underline{1}\ 0\ 0\ 1\ \underline{0}\ 1\ 1\ 0\ 0$.

Now, suppose this code word is sent, and during transmission an error occurs at position 10 so that the receiver gets $\underline{1}\ \underline{1}\ 1\ \underline{1}\ 0\ 0\ 1\ \underline{0}\ 1\ 0\ 0\ 0$. She calculates the parity bits as $\underline{1}\ \underline{0}\ 1\ \underline{1}\ 0\ 0\ 1\ \underline{1}\ 1\ 0\ 0\ 0$ and sees a discrepancy between the received parity and the calculated parity at positions 2 and 8. Because 2 + 8 = 10, she concludes that the bit at position 10 is in error. So, she flips the bit at position 10 to correct the error. The advantage of this correction technique is that the receiver need not compare the received word with all the code words to determine which code word is closest to it.

9.5 RAID Storage Systems

In the early days of computers, disks were physically large and expensive. As technology advanced they became physically small, their data capacities increased, and they became less expensive. They finally got so cheap that it became advantageous to assemble many individual drives into an array of drives, instead of building one bigger drive when large amounts of data needed to be stored. Such a collection is called a *redundant array of inexpensive disks (RAID)* system.

Advantages of RAID systems

The idea is that an array of disks has more spindles, each with its own set of read/write heads that can operate concurrently compared to the single spindle in one big drive. The concurrency should lead to increased performance. Also, redundancy can provide error correction and detection to increase the reliability of the system. The RAID controller provides a level of abstraction to the operating system, making the array of disks appear like one big disk to the operating system. Alternatively, the abstraction can be provided in software as part of the operating system.

There are several different ways to organize an array of disks. The industry-standard terminology for the most common schemes is:

Common RAID levels

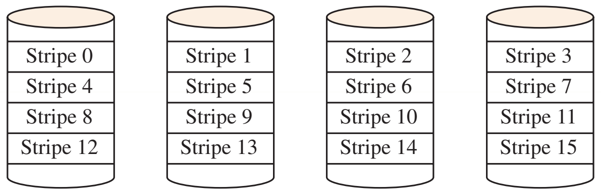
- RAID level 0: Nonredundant striped
- RAID level 1: Mirrored
- RAID levels 01 and 10: Striped and mirrored
- RAID level 2: Memory-style error-correcting code (ECC)
- RAID level 3: Bit-interleaved parity
- RAID level 4: Block-interleaved parity
- RAID level 5: Block-interleaved distributed parity

Each organization has its own set of advantages and disadvantages and is used in different situations. The remainder of this section describes the above RAID levels.

RAID Level 0: Nonredundant Striped

FIGURE 9.26 shows the organization for RAID level 0. Data that would be stored in several contiguous blocks is broken up into stripes and distributed over several disks in the array. Figure 9.18 shows blocks 0 through 7 on one

FIGURE 9.26
RAID level 0: Nonredundant striped.



track, 8 through 15 on the next track, and so on. A stripe consists of several blocks. For example, if there are two blocks per stripe, then blocks 0 and 1 in Figure 9.18 are stored in stripe 0, blocks 2 and 3 in stripe 1, and so on.

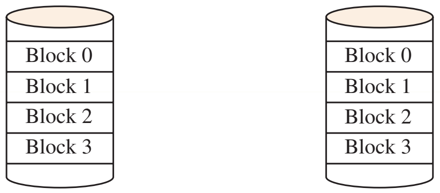
The operating system sees the logical disk as in Figure 9.18, even though the physical disks are as in Figure 9.26. If the operating system requests a disk read of blocks 0 through 7, the RAID system can read stripes 0 through 3 in parallel, decreasing the access time because of the concurrency. To service a read request of blocks 0 through 10—that is, stripes 0 through 5—the first disk would need to deliver stripes 0 and 4 sequentially, as would the second disk with stripes 1 and 5. This organization requires a minimum of two hard drives.

The advantage of level 0 is increased performance. However, it does not work well in an environment where most read/write requests are for a single block or stripe, as there is no concurrency in that case. Also, there is no redundancy as with the other levels, so reliability is not as high. The probability of a single failure with four disks running is greater than the probability of failure of a single disk running, given that all the disks have equal quality.

RAID Level 1: Mirrored

To mirror a disk is to maintain an exact mirror image of it on a separate drive, as shown in FIGURE 9.27. There is no striping, just a strict duplication to provide redundancy in case one of the disk drives fails.

FIGURE 9.27
RAID level 1: Mirrored.



A disk write requires a write to each disk, but they can be done in parallel, so the write performance is not much worse than with a single drive. For a disk read, the controller can choose to read from the drive with the shortest seek time and latency. So a disk read is a bit better than with a single drive. If one drive fails, it can be replaced while the other continues to operate. When the replacement drive is installed, it is easily backed up by duplicating the good drive. Mirroring is usually done with only two drives. If four drives are available, it is generally better to take advantage of the increased performance with striping at level 01.

RAID Levels 01 and 10: Striped and Mirrored

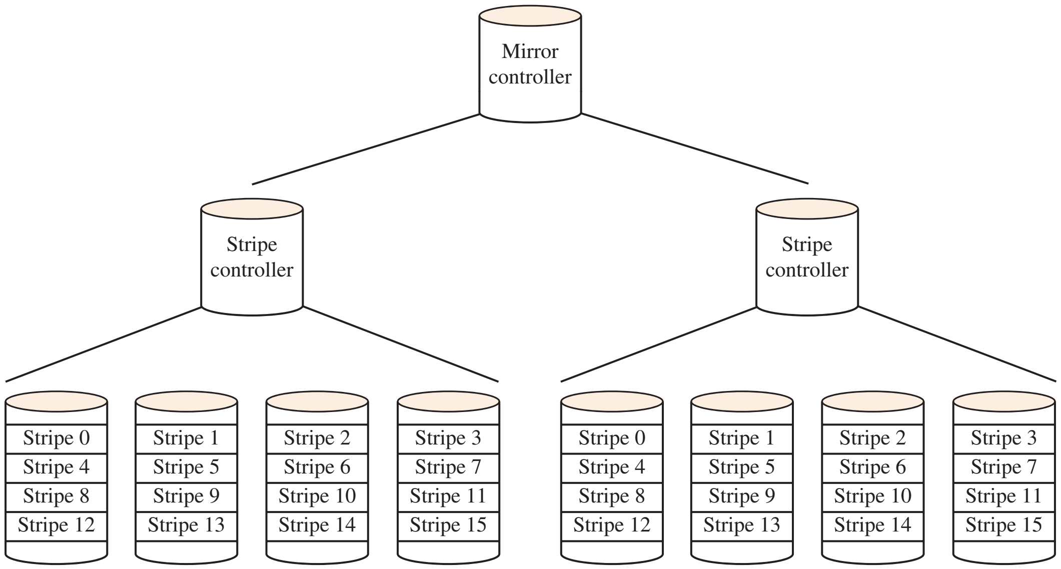
There are two ways to combine RAID levels 0 and 1, and hence to obtain the advantages of both. The first is called RAID level 01, or 0+1, or 0/1, or *mirrored stripes*, as FIGURE 9.28 (a) shows. With mirrored stripes, you simply mirror the disk organization that you would have with striping at level 0. The second is RAID level 10, or 1+0, or 1/0, or *striped mirrors*, as in Figure 9.28(b). Instead of using the redundant disks to duplicate the set of level 0 disks, you mirror pairs of disks, then stripe across the mirrors.

RAID level 10 is more expensive to implement than level 01. With level 01 in Figure 9.28(a), each stripe controller is a system that makes the four striped disks appear as a single disk to the mirror controller. The mirror controller is a system that makes the two mirrored disks appear as a single disk to the computer. With level 10 in Figure 9.28(b), each mirror controller is a system that makes two mirrored disks appear as a single disk to the stripe controller. The stripe controller is a system that makes the four striped disks appear as a single disk to the computer. In this example with eight physical disks, you need only three controllers for level 01 but five controllers for level 10.

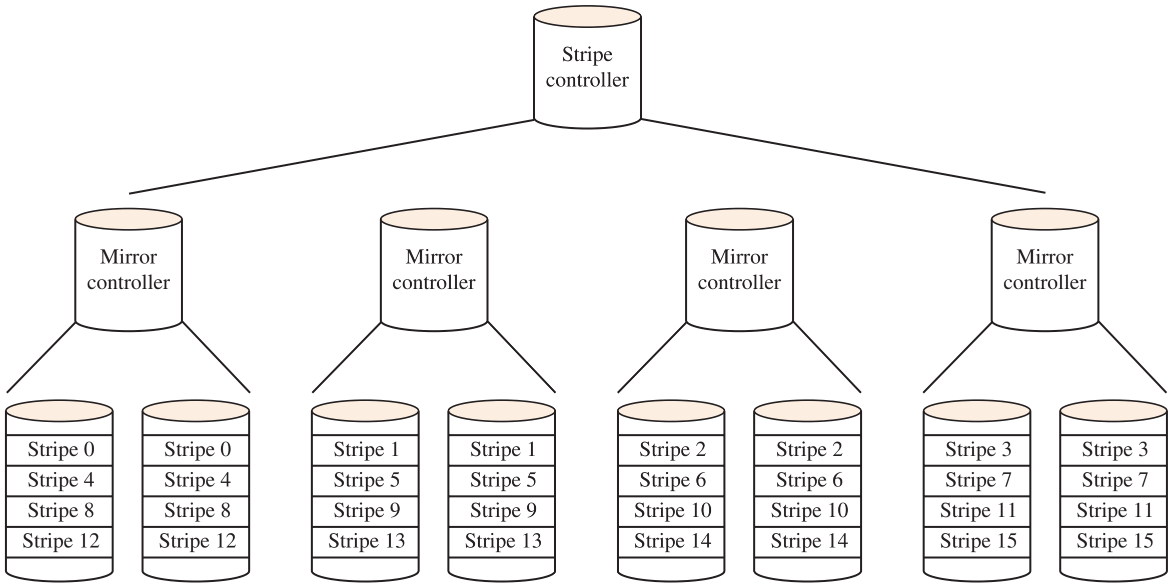
The advantage of level 10 over level 01 is reliability. Suppose one physical disk goes bad, say the third one. In Figure 9.28(a), that bad disk will cause the first stripe controller to report an error to the mirror controller, which will then use its rightmost mirrored disk until the faulty drive can be replaced. In effect, during the downtime, the four physical disks of the left striped disk are out of commission. In Figure 9.28(b), the bad third disk will cause the second mirror controller to use the fourth disk (its second mirrored disk) until the faulty drive can be replaced. During the downtime, only one physical disk is out of commission.

In both cases, the computer sees uninterrupted service from its RAID disk, so it might seem that there is no difference in reliability. However, the problem comes if two disks fail. In Figure 9.28(a), if one of the physical disks fails in the left striped disk and one fails in the right striped disk, the RAID disk fails. If the two disk failures are in the same set of striped disks, the RAID disk does not fail. In Figure 9.28(b), the only way the RAID disk can

FIGURE 9.28
Combining RAID levels 0 and 1.



(a) RAID level 01: Mirrored stripes.



(b) RAID level 10: Striped mirrors.

fail is if both failures are in the same set of paired mirror disks, an event that is less probable than a RAID failure with level 01. See the exercises at the end of this chapter for a quantitative analysis.

Another advantage of level 10 over 01 is the time to do a mirror copy after a failed disk has been replaced. In Figure 9.28(a), the mirror controller sees each striped disk as a single entity, not as four separate disks. Once a repair has been made, the mirror controller has no choice but to copy the entire contents of the good striped disk—that is, four physical disks—to the repaired striped disk. With level 10, all mirrors are with pairs of disks, so only a single disk copy is required to restore a failed disk.

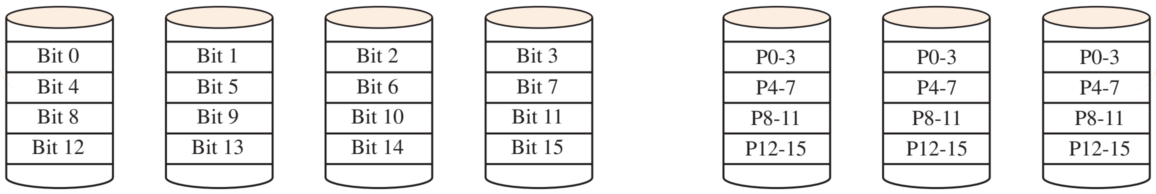
Low-end RAID systems usually support 01, with high-end systems supporting both 01 and 10. You get the performance advantage of striping and the reliability advantage of mirroring. The read performance is even better than level 0 in some cases. Consider the scenario with level 01 of a read request for stripes 0 through 5. Stripes 0 through 3 can be read concurrently on the first set of drives, with stripes 4 and 5 read concurrently on the mirrored set. Both levels 01 and 10 require an even number of hard drives, with a minimum of four.

RAID Level 2: Memory-Style ECC

The storage overhead of mirroring is tremendous—100%—because each drive is duplicated. Figure 9.24 shows that less overhead is possible with single-error-correcting codes as commonly used in high-reliability memory systems. Four data bits can be corrected with three parity bits, bringing the overhead down to 75%. With level 2, you stripe at the bit level. FIGURE 9.29 shows each nybble (half a byte) spread out over the first four drives. The last three drives are the parity bits for the single-error-correcting code.

To maintain performance, the drives must all be rotationally synchronized. To perform a disk write, the disk controller computes the parity bits for each nybble and writes them to the parity drives along with the data. To do a read, the controller computes the parity bits from the data and compares them with the bits from the parity drives, correcting the error on the fly.

FIGURE 9.29
RAID level 2: Memory-style ECC.



This scheme was used on some older supercomputers, usually with 32 data bits and 6 parity bits to get the overhead down. Today, inexpensive drives have their own internal error-correcting capabilities at the bit level, and so level 2 is no longer used commercially.

RAID Level 3: Bit-Interleaved Parity

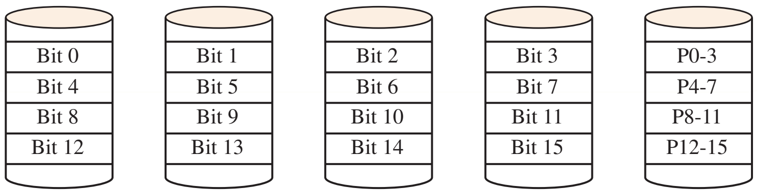
By far the most common failure in a disk array is the failure of just one of the drives in the array. Furthermore, the disk controller can detect such a failure, so the system knows where the failure is. If you stripe at the bit level, and if you know which bit has failed, then you can correct the error with just one parity bit. For example, suppose you want to store the nybble 1001. With even parity, the parity bit is zero, so you store 1001 0. FIGURE 9.30 shows 1001 stored at bit 0, bit 1, bit 2, bit 3, and parity bit 0 stored at P0-3.

Suppose the fourth drive fails, so you know that bits 3, 7, 11, 15, . . . are unavailable. You read your data as 100*x* $\underline{0}$ where *x* is the bit you must correct. Because you are using even parity, you know that the number of 1's must be even and, therefore, that *x* must be 1. Your knowledge of where the error occurred allows you to decrease the overhead for single-error correcting to a single parity bit.

Although level 3 improves on the efficiency of level 2, it has several disadvantages. Recovering from a failed drive is time consuming. With mirroring, you simply clone the content of the one remaining good drive to the replacement drive. With bit-interleaved parity, the bits on the replacement drive must be computed from the bits on all the other drives, which you must, therefore, access. The rebuild is usually done automatically by the controller.

The parity drive is only used to correct errors when a drive fails and to restore the replacement drive. Consequently, it must be written on every write request to update the parity bit. Because individual disk drives have their own ECC at the bit level, you do not access the parity drive on a read request (unless a drive has failed). The access time for level 3 is not much worse than it would be for a single drive. But with levels 2 and 3, every read/

FIGURE 9.30
RAID level 3: Bit-interleaved parity.



write request requires you to access every data drive, so you do not get the concurrency that you do with longer stripes.

RAID Level 4: Block-Interleaved Parity

The only difference between level 3 and level 4 is the size of the stripe. In level 3, a stripe is one bit, and in level 4, it can be one or more blocks. In Figure 9.30, P0-3 represents one bit, but in FIGURE 9.31, P0-3 represents an entire stripe.

For example, if each stripe is 1 KiB long, then a file is distributed over the stripes as follows:

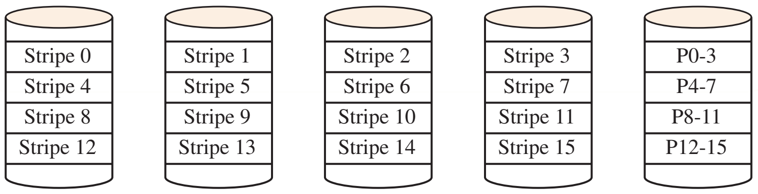
Stripe 0: Bits 0 through 1023
Stripe 1: Bits 1024 through 2047
Stripe 2: Bits 2048 through 3071
Stripe 3: Bits 3072 through 4095

The first bit of P0-3 is the parity bit for bits 0, 1024, 2048, and 3072; the second bit of P0-3 is the parity bit for bits 1, 1025, 2049, and 3073; and so on. Because striping is not at the bit level, disks do not need to be rotationally synchronized, as they do with levels 2 and 3.

Level 4 has an advantage over level 3 with small random read requests. If each file is contained on a few stripes on different disks, the seeks, latencies, and transmissions can all happen concurrently. With level 3, to read even one small file requires all the data drives to act in concert; many small files must be read sequentially.

Although overhead with level 4 is much reduced compared to mirrored organizations, its biggest drawback is with write requests. If you are writing a file that spans stripes 0 through 3, you can compute the parity for P0-3 and write it to the parity drive at the same time. But suppose you need to write a file that is wholly contained in stripe 0. Because you are going to alter stripe 0, you also must alter P0-3. But P0-3 is the parity for stripes 1, 2, and 3 as well as 0. It would seem that you have to read stripes 1, 2, and 3 to use with your new stripe 0 to compute the new parity. There is a more efficient way,

FIGURE 9.31
RAID level 4: Block-interleaved parity.



however. Instead of reading stripes 1, 2, and 3, you can read the old stripe 0 and the old P0-3. For each bit position in the new and old data stripes, if your new bit is different from your old bit, then you will be changing the number of 1's from an even number to an odd number or vice versa, and you must flip that bit in P0-3. If the new and old data bits are the same, you leave the corresponding parity bit unchanged. For four data disks, this technique reduces the number of disk reads from three to two.

Even with this shortcut, every write request requires a write to the parity disk, no matter how small the request. The parity disk becomes the performance bottleneck.

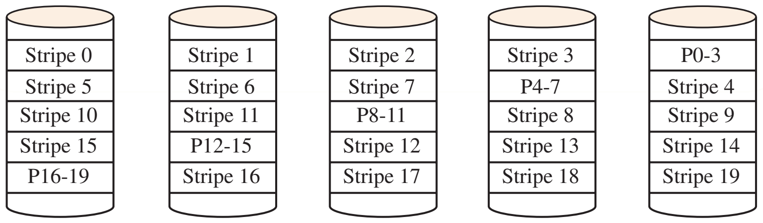
RAID Level 5: Block-Interleaved Distributed Parity

Level 5 alleviates the parity disk bottleneck. Rather than store all the parity on one disk, the parity information is scattered among all the disks, so that no one disk has the responsibility for the parity information of the whole array.

FIGURE 9.32 shows a common organization, known as *left-symmetric parity distribution*, for spreading the parity information among all the disks. It has the advantage that if you read a set of stripes sequentially, you access each disk once before accessing any disk twice. In the figure, suppose you access stripes 0, 1, 2, 3, and 4 in that order. You will access the first, second, third, fourth, and fifth disks. If you put stripe 4 where stripe 5 is in the figure and service the same request, you would access the first, second, third, fourth, and first disk; that is, you would access the first disk twice before accessing the fifth disk at all. You can see that the desirable property holds regardless of which stripe you begin with.

RAID level 5 is considered by many to be the ideal combination of good reliability, good performance, high capacity, and low storage overhead. It is one of the most popular high-end RAID systems. The most popular low-end system is probably RAID level 0, which is not really a true RAID because there is no redundancy, and hence no enhanced reliability.

FIGURE 9.32
RAID level 5: Block-interleaved distributed parity.



Chapter Summary

The operating system allocates time in the form of CPU utilization and space in the form of main memory and disk allocation. Five techniques for allocation of main memory are uniprogramming, fixed-partition multiprogramming, variable-partition multiprogramming, paging, and virtual memory. With uniprogramming, only one job executes at a time from start to finish, and the job has the entire main memory to itself. Fixed-partition multiprogramming allows several jobs to execute concurrently and requires the operating system to determine the partition sizes in memory before executing any jobs. Variable-partition multiprogramming alleviates the inefficiencies inherent in fixed-partition multiprogramming by allowing the partition sizes to vary depending on the job requirements. The best-fit and first-fit algorithms are two different strategies to cope with the fragmentation problem of variable-partition multiprogramming.

Paging alleviates fragmentation by fragmenting the program to fit the memory holes. Programs are no longer contiguous but are broken up and scattered throughout main memory. Jobs are divided into equal-sized pages, and main memory is divided into frames of the same size. Logical addresses as seen by the programmer are converted to physical addresses with the help of a page table. The page table contains the frame number for each page that is stored in main memory.

Demand paging, also called *virtual memory*, postpones page loading into memory until the job demands the page. The entire program does not need to reside in main memory to execute. Instead, only its active pages, called the *working set*, are loaded. A page fault occurs when a page is referenced but has not yet been loaded into memory. First-in, first-out (FIFO) and least recently used (LRU) are two algorithms for determining which page to swap out of main memory when a frame is needed for a new page. You would normally expect the number of faults to decrease with increasing number of memory frames. But Bélády's anomaly shows that it is possible for an increase in the number of frames to produce an increase in the number of faults with the FIFO replacement algorithm.

Three contributions to disk access time are seek time, which is the time it takes for the arm to move to the designated cylinder; latency, which is the time it takes for the block to rotate to the head once the head is in place; and transmission time, which is the time it takes for the block to pass underneath the head. Three techniques of disk management are contiguous, linked, and indexed. The problem of fragmentation occurs with disk memory, as it does with main memory.

A set of redundant bits can be added to data bits in order to detect or correct errors that may occur during transmission or storage of data.

The Hamming distance between two code words is the number of bits that are different. The receiver corrects errors by choosing the code word that is closest to the received word based on the Hamming distance. With a judicious choice of the placement of the redundant bits, you can correct a single error without comparing the received word with all the code words.

A redundant array of inexpensive disks (RAID) is a grouping of disks that appears to the operating systems as a single large disk. The two benefits of a RAID system are performance, based on the concurrent access of the data with multiple spindles in the system, and reliability, based on error correction and detection with redundant drives.

Exercises

Section 9.1

1. Using the format of Figure 9.4, devise a job execution sequence for which the first-fit algorithm would require compaction before the best-fit algorithm. Sketch the fragmentation in main memory just before compaction is required for each algorithm.

2. Figure 9.10 shows how a page table in a paging system performs the same transformation of the logical address as the base register does in a multiprogramming system. The equivalent job of the bound register is not shown in the figure. ***(a)** To protect other processes' memory space from unauthorized access, would a paging system require a table of bound values, one for each page, or would a single bound register suffice? Explain. **(b)** Modify Figure 9.10 to include main memory protection from other processes.

3. Suppose the page size in a paging system is 512 bytes. **(a)** If most of the files are large—that is, much greater than 512 bytes—what do you suppose is the average internal fragmentation (in bytes of unused space) for each file? Explain your reasoning. **(b)** How would your answer to part (a) change if most of the files were much smaller than 512 bytes? **(c)** How would your answer to part (b) change if you expressed the fragmentation in terms of the percentage of unused space instead of the number of unused bytes?

Section 9.2

4. A computer has 12-bit addresses and a main memory that is divided into 16 frames. Memory management uses demand paging. ***(a)** How many bytes is virtual memory? **(b)** How many bytes are in each page? **(c)** How many bits are in the offset of a logical and physical address? **(d)** What is the maximum number of entries in a job's page table?

5. Answer Exercise 4 for a computer with *n*-bit addresses and a memory divided into 2^k frames.

*__6.__ For which pages in Figure 9.12 is the image on disk an exact replica of the page in main memory?

7. Verify the data of Figure 9.15(b), which shows Bélády's anomaly, for the sequence of page references given in the text. Display the content of the frames in the format of Figure 9.13.

*__8.__ Devise a sequence of 12 page references for which the FIFO page-replacement algorithm is better than the LRU algorithm.

9. Plot the graph of Figure 9.15(b) for the page reference sequence in Figure 9.13 using the FIFO page-replacement algorithm. On the same graph, plot the data for the LRU algorithm.

*__10.__ If the operating system could predict the future, it could select the replacement page such that the next page fault is delayed as long as possible. Such an algorithm is called *OPT*, the *optimum page-replacement algorithm*. It is a useful theoretical algorithm because it represents the best you could possibly do. When designers measure the performance of their page-replacement algorithms, they try to get as close as possible to the performance of OPT. How many page faults does OPT produce for the sequence of Figures 9.13 and 9.16? How does that compare with FIFO and LRU?

Section 9.3

11. Suppose a disk rotates at 5,400 revolutions per minute and has each surface divided into 16 sectors. ***(a)** What is the maximum possible latency time? Under what circumstance will that occur? **(b)** What is the minimum possible latency time? Under what circumstance will that occur? **(c)** From (a) and (b), what will be the average latency time? **(d)** What is the transmission time for one block?

Section 9.4

12. ***(a)** How many data bits are required to store one of the decimal digits 0 through 9? ***(b)** How many parity bits are required to *detect* a single error? **(c)** Write a single-error detection code using even parity. Underline the parity bits. **(d)** What is the code distance of your code?

13. **(a)** What must the code distance be to *detect* five errors? **(b)** What must the code distance be to *correct* five errors?

14. **(a)** Which entries in Figure 9.24 represent perfect codes? **(b)** Augment the table in Figure 9.24 with additional entries to include all the perfect codes between $m = 4$ and $m = 128$. Be sure to include the overhead value. **(c)** What can you conclude about the cost of restricting the number of data bits to a power of 2?

15. **(a)** How many data bits are required to store one of the decimal digits 0 through 9? **(b)** How many parity bits are required to *correct* a single error? **(c)** Write a single-error correction code using even parity. Underline the parity bits. **(d)** What is the code distance of your code?

16. A set of eight data bits is transmitted with the single-error correction code of Figure 9.25. For each of the received bit patterns below, state whether an error occured. If it did, correct the error.

*__(a)__ 1 0 0 1 1 0 1 0 1 0 0 1 **(b)** 1 1 0 1 0 0 1 1 0 0 1 0
(c) 0 0 0 0 1 0 1 1 0 1 0 0 **(d)** 1 0 1 1 0 0 1 0 0 1 0 0

Section 9.5

17. Figure 9.28 shows a RAID system with eight physical disks. **(a)** With six physical disks, how many mirror controllers and stripe controllers would you need for level 01 and for level 10? **(b)** With $2n$ disks in general (so that $n = 4$ in Figure 9.28), how many mirror controllers and stripe controllers would you need for level 01 and for level 10?

18. **(a)** Figure 9.28 shows the RAID level 01 and RAID level 10 systems with eight physical disks. Draw the equivalent systems for level 01 and level 10 with four physical disks. **(b)** Assume that two disks go bad. The sequence BBGG means that the first and second disks are bad and the third and fourth disks are good. With this scenario, the RAID level 01 disk is good because the two bad disks are in the same first striped disk, but the RAID level 10 disk is bad because the two bad disks are in the same first mirrored disk. How many permutations of four letters with two B's and two G's are there? **(c)** Tabulate each permutation, and for each one determine whether the RAID disk is good or bad for levels 01 and 10. **(d)** If two disks fail, use part (c) to determine the probability that the RAID disk fails for levels 01 and 10. Which RAID system is more reliable? **(e)** With $2n$ disks in general (so that $n = 4$ in Figure 9.28), how many permutations of $2n$ letters are there with 2 B's and $2n - 2$ G's? **(f)** How many of the permutations from part (e) cause a RAID disk failure for level 01 and for level 10? **(g)** If two disks fail, use part (f) to determine the probability that the RAID disk fails for levels 01 and 10. **(h)** Use part (g) to show that the probability that the RAID disk in Figure 9.28 fails is 4/7 for level 01 and 1/7 for level 10 if two disks fail.

19. You have a RAID level 4 system with eight data disks and one parity disk. **(a)** How many disk reads and disk writes must you make to write one data stripe if you do not make use of the old data and parity values? **(b)** How many disk reads and disk writes must you make to write one data stripe if you do make use of the old data and parity values?