OS_CN_Compiler

Assembled by Jainish Parmar

Table of Contents

Wiley Loose-Leaf Print Edition

Operating System Concepts

TENTH EDITION

ABRAHAM SILBERSCHATZ • PETER BAER GALVIN • GREG GAGNE

With End of Chapter Exercises

WILEY

CHAPTER 10

Virtual Memory

In Chapter 9, we discussed various memory-management strategies used in computer systems. All these strategies have the same goal: to keep many processes in memory simultaneously to allow multiprogramming. However, they tend to require that an entire process be in memory before it can execute.

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the programmer from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to share files and libraries, and to implement shared memory. In addition, it provides an efficient mechanism for process creation. Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly. In this chapter, we provide a detailed overview of virtual memory, examine how it is implemented, and explore its complexity and benefits.

CHAPTER OBJECTIVES

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process, and explain how it is related to program locality.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.
- Design a virtual memory manager simulation in the C programming language.

10.1 Background

The memory-management algorithms outlined in Chapter 9 are necessary because of one basic requirement: the instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Dynamic linking can help to ease this restriction, but it generally requires special precautions and extra work by the programmer.

The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance, consider the following:

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
- Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have not been used in many years.

Even in those cases where the entire program is needed, it may not all be needed at the same time.

The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large ***virtual*** address space, simplifying the programming task.
- Because each program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system and its users.

Virtual memory involves the separation of logical memory as perceived by developers from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 10.1). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on programming the problem that is to be solved.

The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address—say, address 0—and exists in contiguous memory, as shown in Figure 10.2. Recall from Chapter 9, though, that in fact physical memory is organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory-

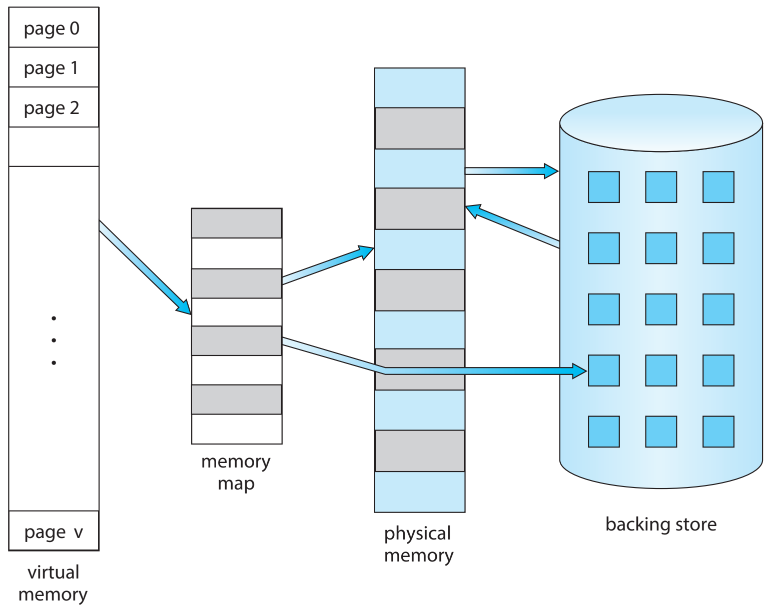


Figure 10.1 Diagram showing virtual memory that is larger than physical memory.

management unit (MMU) to map logical pages to physical page frames in memory.

Note in Figure 10.2 that we allow the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as **sparse** address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

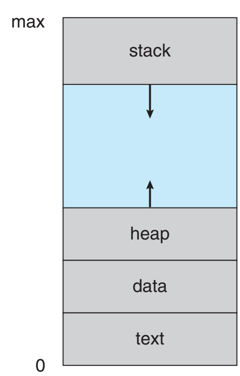


Figure 10.2 Virtual address space of a process in memory.

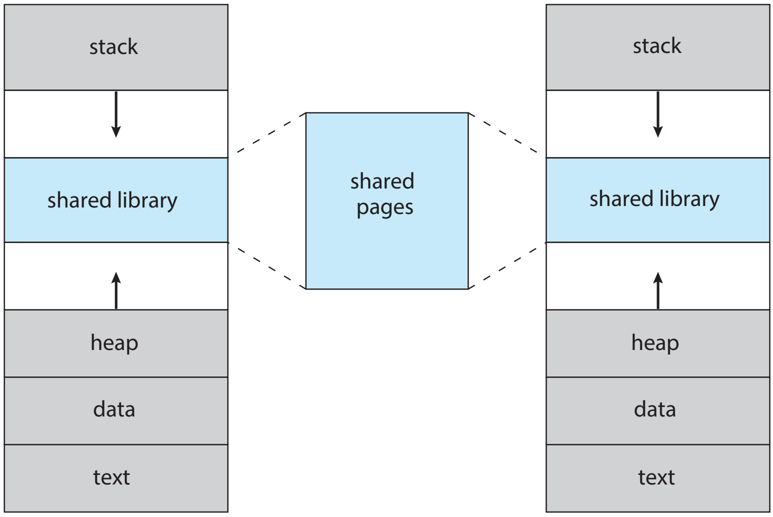


Figure 10.3 Shared library using virtual memory.

In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing (Section 9.3.4). This leads to the following benefits:

- System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes (Figure 10.3). Typically, a library is mapped read-only into the space of each process that is linked with it.
- Similarly, processes can share memory. Recall from Chapter 3 that two or more processes can communicate through the use of shared memory. Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in Figure 10.3.
- Pages can be shared during process creation with the `fork()` system call, thus speeding up process creation.

We further explore these—and other—benefits of virtual memory later in this chapter. First, though, we discuss implementing virtual memory through demand paging.

10.2 Demand Paging

Consider how an executable program might be loaded from secondary storage into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially ***need*** the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for ***all*** options, regardless of whether or not an option is ultimately selected by the user.

An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are loaded only when they are ***demanded*** during program execution. Pages that are never accessed are thus never loaded into physical memory. A demand-paging system is similar to a paging system with swapping (Section 9.5.2) where processes reside in secondary memory (usually an HDD or NVM device). Demand paging explains one of the primary benefits of virtual memory—by loading only the portions of programs that are needed, memory is used more efficiently.

10.2.1 Basic Concepts

The general concept behind demand paging, as mentioned, is to load a page in memory only when it is needed. As a result, while a process is executing, some pages will be in memory, and some will be in secondary storage. Thus, we need some form of hardware support to distinguish between the two. The valid–invalid bit scheme described in Section 9.3.3 can be used for this purpose. This

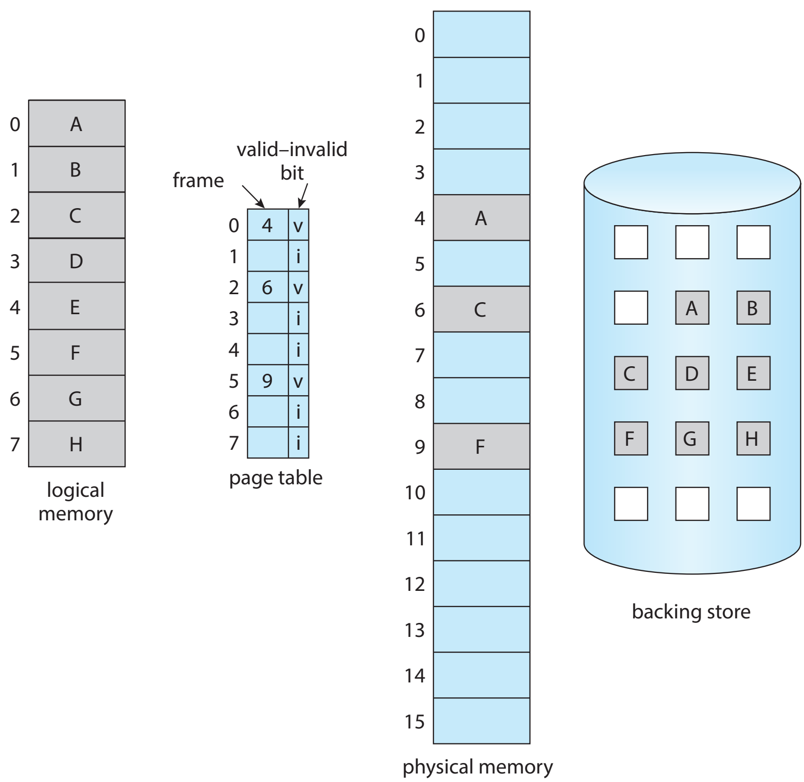


Figure 10.4 Page table when some pages are not in main memory.

time, however, when the bit is set to “valid,” the associated page is both legal and in memory. If the bit is set to “invalid,” the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently in secondary storage. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid. This situation is depicted in Figure 10.4. (Notice that marking a page invalid will have no effect if the process never attempts to access that page.)

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page fault**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (Figure 10.5):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).

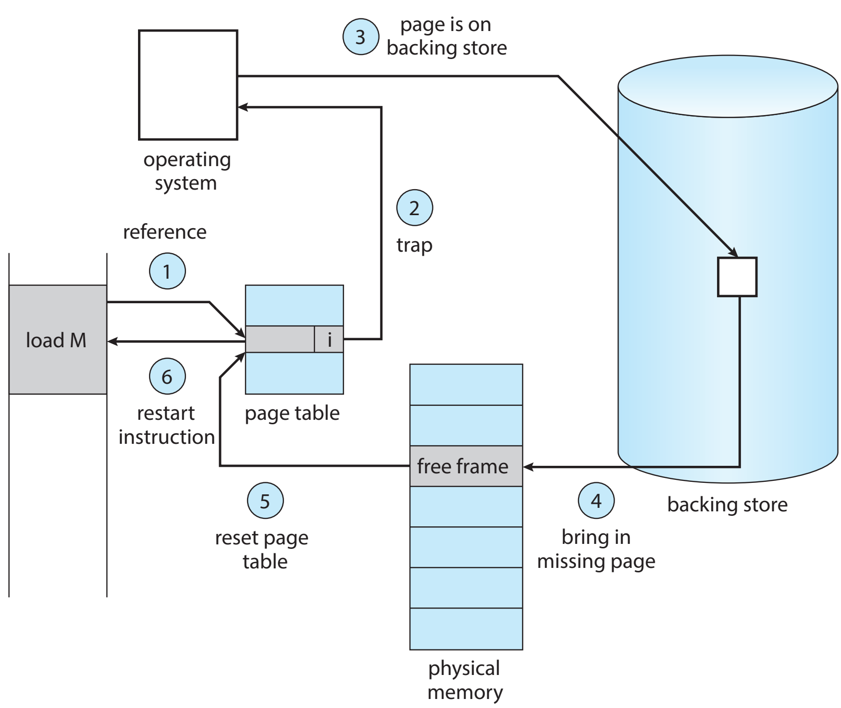


Figure 10.5 Steps in handling a page fault.

4. We schedule a secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with ***no*** pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is **pure demand paging**: never bring a page into memory until it is required.

Theoretically, some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behavior is exceedingly unlikely. Programs tend to have **locality of reference**, described in Section 10.6.1, which results in reasonable performance from demand paging.

The hardware to support demand paging is the same as the hardware for paging and swapping:

- **Page table**. This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.
- **Secondary memory**. This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk or NVM device. It is known as the swap device, and the section of storage used for this purpose is known as **swap space**. Swap-space allocation is discussed in Chapter 11.

A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in ***exactly*** the same place and state, except that the desired page is now in memory and is accessible. In most cases, this requirement is easy to meet. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

As a worst-case example, consider a three-address instruction such as ADD the content of A to B, placing the result in C. These are the steps to execute this instruction:

1. Fetch and decode the instruction (ADD).
2. Fetch A.

3. Fetch B.
4. Add A and B.
5. Store the sum in C.

If we fault when we try to store in C (because C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction. The restart will require fetching the instruction again, decoding it again, fetching the two operands again, and then adding again. However, there is not much repeated work (less than one complete instruction), and the repetition is necessary only when a page fault occurs.

The major difficulty arises when one instruction may modify several different locations. For example, consider the IBM System 360/370 MVC (move character) instruction, which can move up to 256 bytes from one location to another (possibly overlapping) location. If either block (source or destination) straddles a page boundary, a page fault might occur after the move is partially done. In addition, if the source and destination blocks overlap, the source block may have been modified, in which case we cannot simply restart the instruction.

This problem can be solved in two different ways. In one solution, the microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified. The move can then take place; we know that no page fault can occur, since all the relevant pages are in memory. The other solution uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs. This action restores memory to its state before the instruction was started, so that the instruction can be repeated.

This is by no means the only architectural problem resulting from adding paging to an existing architecture to allow demand paging, but it illustrates some of the difficulties involved. Paging is added between the CPU and the memory in a computer system. It should be entirely transparent to a process. Thus, people often assume that paging can be added to any system. Although this assumption is true for a non-demand-paging environment, where a page fault represents a fatal error, it is not true where a page fault means only that an additional page must be brought into memory and the process restarted.

10.2.2 Free-Frame List

When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory. To resolve page faults, most operating systems maintain a **free-frame list**, a pool of free frames for satisfying such requests (Figure 10.6). (Free frames must also be allocated when the stack or heap segments from a process expand.) Operating systems typically allo-

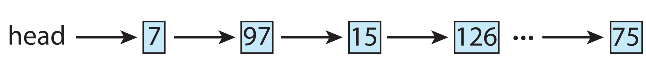


Figure 10.6 List of free frames.

cate free frames using a technique known as **zero-fill-on-deman** . Zero-fill-on-demand frames are "zeroed-out" before being allocated, thus erasing their previous contents. (Consider the potential security implications of ***not*** clearing out the contents of a frame before reassigning it.)

When a system starts up, all available memory is placed on the free-frame list. As free frames are requested (for example, through demand paging), the size of the free-frame list shrinks. At some point, the list either falls to zero or falls below a certain threshold, at which point it must be repopulated. We cover strategies for both of these situations in Section 10.4.

10.2.3 Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the **effective access time** for a demand-paged memory. Assume the memory-access time, denoted *ma*, is 10 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from secondary storage and then access the desired word.

Let p be the probability of a page fault ($0 \le p \le 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults. The effective access time is then

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time}.$$

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal, and determine the location of the page in secondary storage.
5. Issue a read from the storage to a free frame:
    a. Wait in a queue until the read request is serviced.
    b. Wait for the device seek and/or latency time.
    c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU core to some other process.
7. Receive an interrupt from the storage I/O subsystem (I/O completed).
8. Save the registers and process state for the other process (if step 6 is executed).
9. Determine that the interrupt was from the secondary storage device.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU core to be allocated to this process again.

12. Restore the registers, process state, and new page table, and then resume the interrupted instruction.

Not all of these steps are necessary in every case. For example, we are assuming that, in step 6, the CPU is allocated to another process while the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization but requires additional time to resume the page-fault service routine when the I/O transfer is complete.

In any case, there are three major task components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. Let's consider the case of HDDs being used as the paging device. The page-switch time will probably be close to 8 milliseconds. (A typical hard disk has an average latency of 3 milliseconds, a seek of 5 milliseconds, and a transfer time of 0.05 milliseconds. Thus, the total paging time is about 8 milliseconds, including hardware and software time.) Remember also that we are looking at only the device-service time. If a queue of processes is waiting for the device, we have to add queuing time as we wait for the paging device to be free to service our request, increasing even more the time to page in.

With an average page-fault service time of 8 milliseconds and a memory-access time of 200 nanoseconds, the effective access time in nanoseconds is

$$\begin{aligned}\text{effective access time} &= (1 - p) \times (200) + p\ (8\ \text{milliseconds})\\ &= (1 - p) \times 200 + p \times 8{,}000{,}000\\ &= 200 + 7{,}999{,}800 \times p.\end{aligned}$$

We see, then, that the effective access time is directly proportional to the **page-fault rate**. If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds. The computer will be slowed down by a factor of 40 because of demand paging! If we want performance degradation to be less than 10 percent, we need to keep the probability of page faults at the following level:

$$\begin{aligned}&220 > 200 + 7{,}999{,}800 \times p,\\ &20 > 7{,}999{,}800 \times p,\\ &p < 0.0000025.\end{aligned}$$

That is, to keep the slowdown due to paging at a reasonable level, we can allow fewer that one memory access out of 399,990 to page-fault. In sum, it is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

An additional aspect of demand paging is the handling and overall use of swap space. I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used (Chapter 11). One option for the system to gain better paging throughput is by copying an entire file image into the swap space at process startup and then performing demand paging from the swap space. The obvious disadvantage of this approach is the copying of the file image at program start-up. A second option—and one practiced by several operating systems, including Linux and Windows—is to demand-page from the file system initially but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are read from the file system but that all subsequent paging is done from swap space.

Some systems attempt to limit the amount of swap space used through demand paging of binary executable files. Demand pages for such files are brought directly from the file system. However, when page replacement is called for, these frames can simply be overwritten (because they are never modified), and the pages can be read in from the file system again if needed. Using this approach, the file system itself serves as the backing store. However, swap space must still be used for pages not associated with a file (known as **anonymous memory**); these pages include the stack and heap for a process. This method appears to be a good compromise and is used in several systems, including Linux and BSD UNIX.

As described in Section 9.5.3, mobile operating systems typically do not support swapping. Instead, these systems demand-page from the file system and reclaim read-only pages (such as code) from applications if memory becomes constrained. Such data can be demand-paged from the file system if it is later needed. Under iOS, anonymous memory pages are never reclaimed from an application unless the application is terminated or explicitly releases the memory. In Section 10.7, we cover compressed memory, a commonly used alternative to swapping in mobile systems.

10.3 Copy-on-Write

In Section 10.2, we illustrated how a process can start quickly by demand-paging in the page containing the first instruction. However, process creation using the `fork()` system call may initially bypass the need for demand paging by using a technique similar to page sharing (covered in Section 9.3.4). This technique provides rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.

Recall that the `fork()` system call creates a child process that is a duplicate of its parent. Traditionally, `fork()` worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the `exec()` system call immediately after creation, the copying of the parent's address space may be unnecessary. Instead, we can use a technique known as **copy-on-write**, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. Copy-on-write is illustrated in Figures 10.7 and 10.8, which show the contents of the physical memory before and after process 1 modifies page C.

For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write. The operating system will obtain a frame from the free-frame list and create a copy

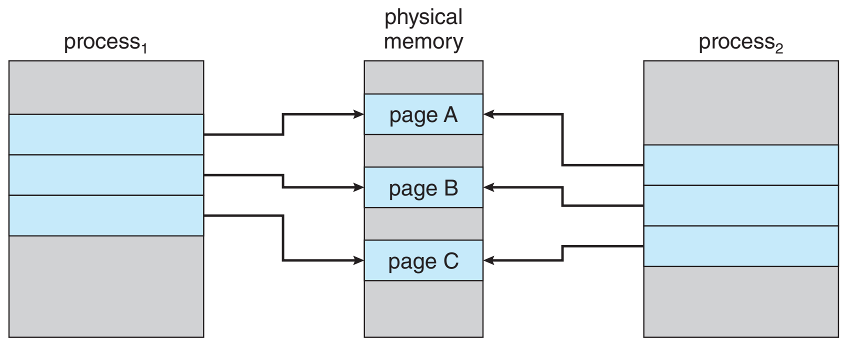


Figure 10.7 Before process 1 modifies page C.

of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process. Obviously, when the copy-on-write technique is used, only the pages that are modified by either process are copied; all unmodified pages can be shared by the parent and child processes. Note, too, that only pages that can be modified need be marked as copy-on-write. Pages that cannot be modified (pages containing executable code) can be shared by the parent and child. Copy-on-write is a common technique used by several operating systems, including Windows, Linux, and macOS.

Several versions of UNIX (including Linux, macOS, and BSD UNIX) provide a variation of the `fork()` system call—`vfork()` (for **virtual memory fork**)—that operates differently from `fork()` with copy-on-write. With `vfork()`, the parent process is suspended, and the child process uses the address space of the parent. Because `vfork()` does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes. Therefore, `vfork()` must be used with caution to ensure that the child process does not modify the address space of the parent. `vfork()` is intended to be used when the child process calls `exec()` immediately after creation. Because no copying of pages takes place, `vfork()`

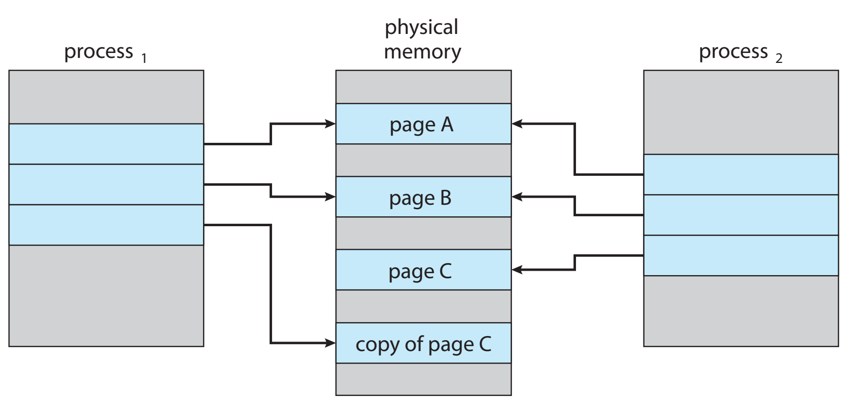


Figure 10.8 After process 1 modifies page C.

is an extremely efficient method of process creation and is sometimes used to implement UNIX command-line shell interfaces.

10.4 Page Replacement

In our earlier discussion of the page-fault rate, we assumed that each page faults at most once, when it is first referenced. This representation is not strictly accurate, however. If a process of ten pages actually uses only half of them, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had forty frames, we could run eight processes, rather than the four that could run if each required ten frames (five of which were never used).

If we increase our degree of multiprogramming, we are **over-allocating** memory. If we run six processes, each of which is ten pages in size but actually uses only five pages, we have higher CPU utilization and throughput, with ten frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for sixty frames when only forty are available.

Further, consider that system memory is not used only for holding program pages. Buffers for I/O also consume a considerable amount of memory. This use can increase the strain on memory-placement algorithms. Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge. Some systems allocate a fixed percentage of memory for I/O buffers, whereas others allow both processes and the I/O subsystem to compete for all system memory. Section 14.6 discusses the integrated relationship between I/O buffers and virtual memory techniques.

Over-allocation of memory manifests itself as follows. While a process is executing, a page fault occurs. The operating system determines where the desired page is residing on secondary storage but then finds that there are ***no*** free frames on the free-frame list; all memory is in use. This situation is illustrated in Figure 10.9, where the fact that there are no free frames is depicted by a question mark.

The operating system has several options at this point. It could terminate the process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system—paging should be logically transparent to the user. So this option is not the best choice.

The operating system could instead use standard swapping and swap out a process, freeing all its frames and reducing the level of multiprogramming. However, as discussed in Section 9.5, standard swapping is no longer used by most operating systems due to the overhead of copying entire processes between memory and swap space. Most operating systems now combine swapping pages with **page replacement**, a technique we describe in detail in the remainder of this section.

10.4.1 Basic Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its

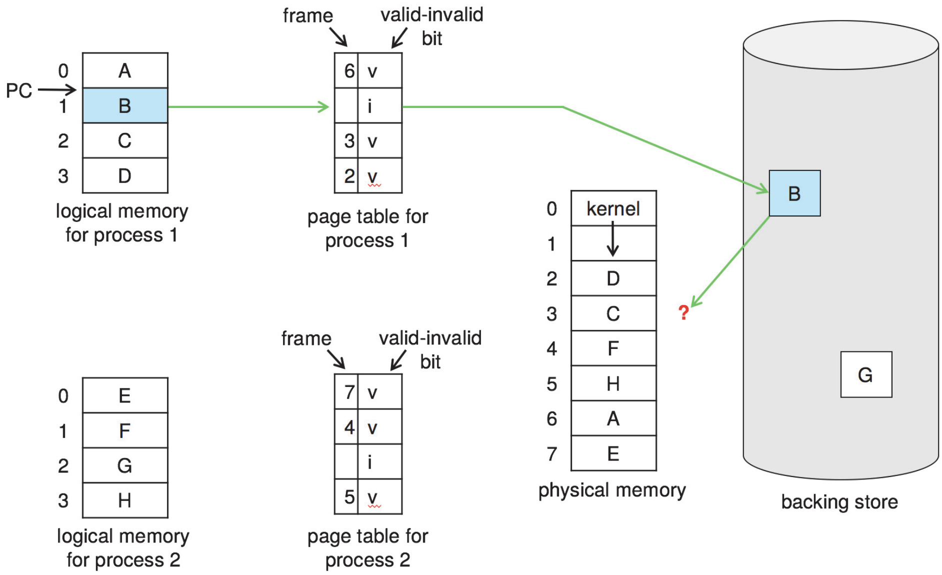


Figure 10.9 Need for page replacement.

contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 10.10). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on secondary storage.
2. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page-replacement algorithm to select a **victim frame**.
   c. Write the victim frame to secondary storage (if necessary); change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the process from where the page fault occurred.

Notice that, if no frames are free, *two* page transfers (one for the page-out and one for the page-in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

We can reduce this overhead by using a **modify bit** (or **dirty bit**). When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set,

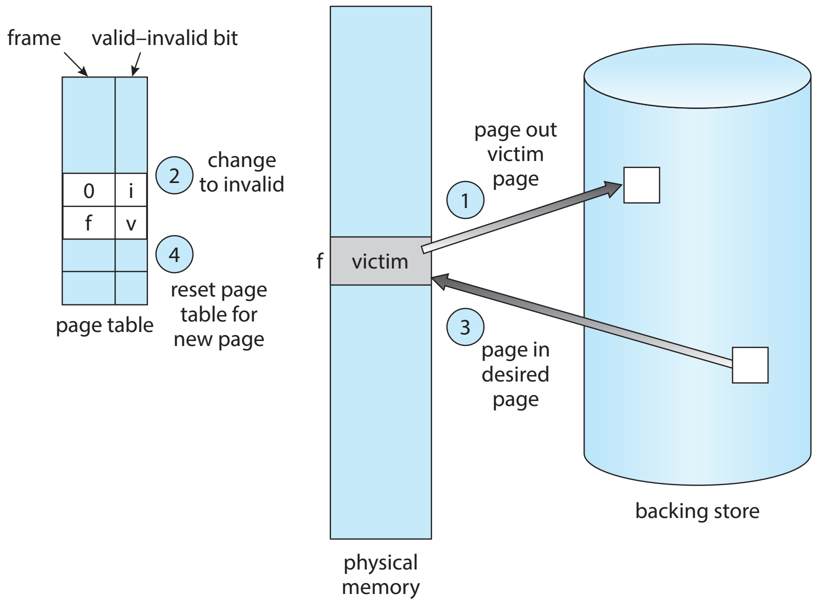


Figure 10.10 Page replacement.

we know that the page has been modified since it was read in from secondary storage. In this case, we must write the page to storage. If the modify bit is not set, however, the page has ***not*** been modified since it was read into memory. In this case, we need not write the memory page to storage: it is already there. This technique also applies to read-only pages (for example, pages of binary code). Such pages cannot be modified; thus, they may be discarded when desired. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half ***if*** the page has not been modified.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. With no demand paging, logical addresses are mapped into physical addresses, and the two sets of addresses can be different. All the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a process of twenty pages, we can execute it in ten frames simply by using demand paging and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to secondary storage. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process.

We must solve two major problems to implement demand paging: we must develop a **frame-allocation algorithm** and a **page-replacement algorithm**. That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because secondary storage

I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.

There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**. We can generate reference strings artificially (by using a random-number generator, for example), or we can trace a given system and record the address of each memory reference. The latter choice produces a large number of data (on the order of 1 million addresses per second). To reduce the number of data, we use two facts.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address. Second, if we have a reference to a page p, then any references to page p that ***immediately*** follow will never cause a page fault. Page p will be in memory after the first reference, so the immediately following references will not fault.

For example, if we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

At 100 bytes per page, this sequence is reduced to the following reference string:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults decreases. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults—one fault for the first reference to each page. In contrast, with only one frame available, we would have a replacement with every reference, resulting in eleven faults. In general, we expect a curve such as that in Figure 10.11. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

We next illustrate several page-replacement algorithms. In doing so, we use the reference string

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

for a memory with three frames.

10.4.2 FIFO Page Replacement

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a

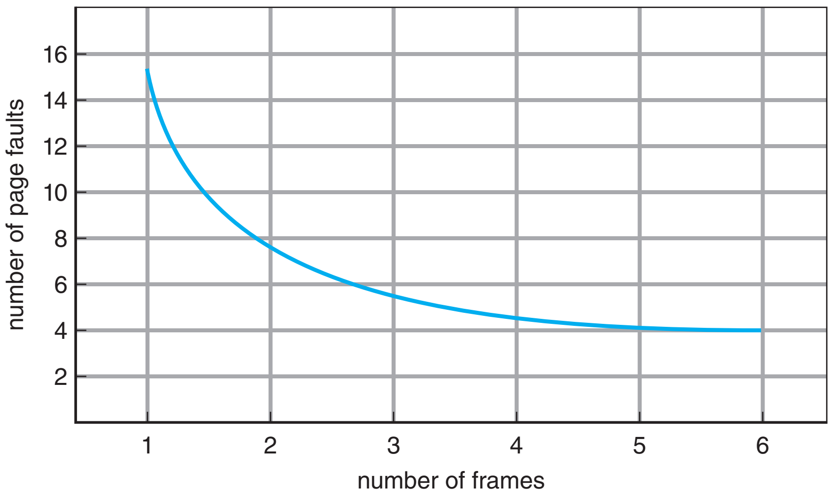


Figure 10.11 Graph of page faults versus number of frames.

page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 10.12. Every time a fault occurs, we show which pages are in our three frames. There are fifteen faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new

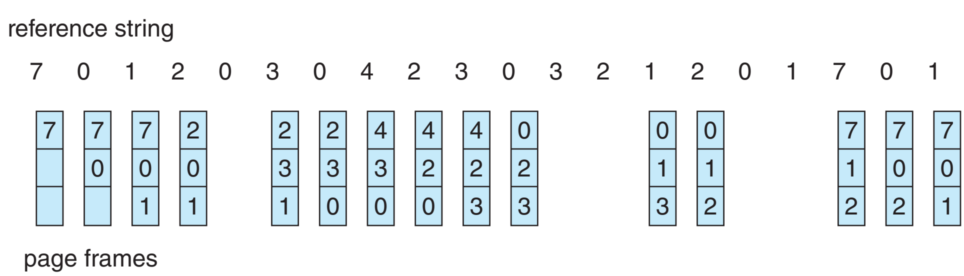


Figure 10.12 FIFO page-replacement algorithm.

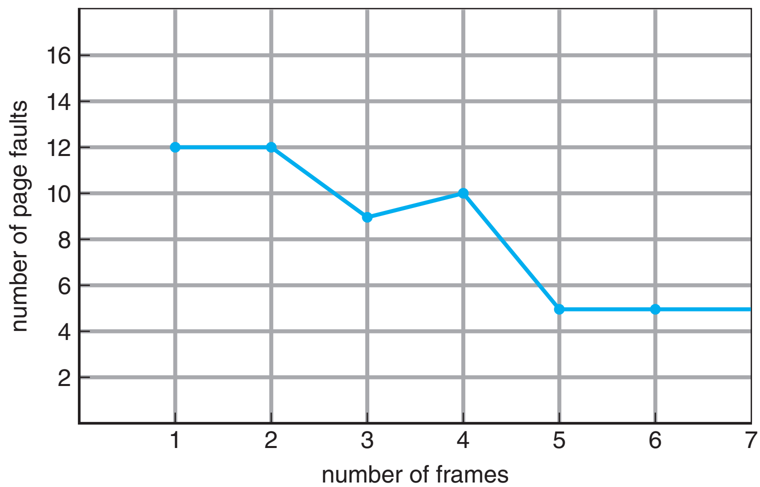


Figure 10.13 Page-fault curve for FIFO replacement on a reference string.

one, a fault occurs almost immediately to retrieve the active page. Some other page must be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution. It does not, however, cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, consider the following reference string:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Figure 10.13 shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is ***greater*** than the number of faults for three frames (nine)! This most unexpected result is known as **Belady's anomaly**: for some page-replacement algorithms, the page-fault rate may ***increase*** as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

10.4.3 Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an **optimal page-replacement algorithm**—the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this:

Replace the page that will not be used for the longest period of time.

Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 10.14. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas

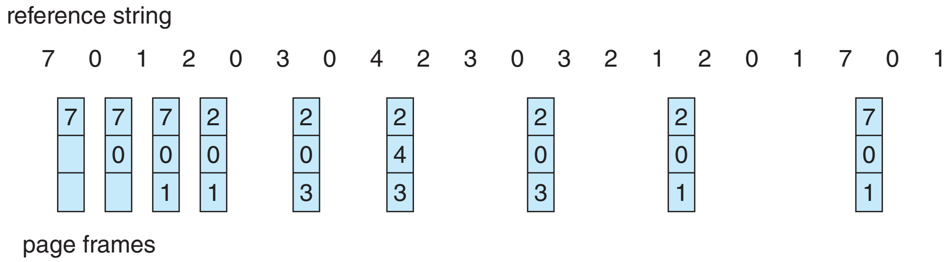


Figure 10.14 Optimal page-replacement algorithm.

page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (We encountered a similar situation with the SJF CPU-scheduling algorithm in Section 5.3.2.) As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be useful to know that, although a new algorithm is not optimal, it is within 12.3 percent of optimal at worst and within 4.7 percent on average.

10.4.4 LRU Page Replacement

If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be ***used.*** If we use the recent past as an approximation of the near future, then we can replace the page that ***has not been used*** for the longest period of time. This approach is the **least recently used (LRU) algorithm**.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward. (Strangely, if we let S^R be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on S^R. Similarly, the page-fault rate for the LRU algorithm on S is the same as the page-fault rate for the LRU algorithm on S^R.)

The result of applying LRU replacement to our example reference string is shown in Figure 10.15. The LRU algorithm produces twelve faults. Notice that the first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults

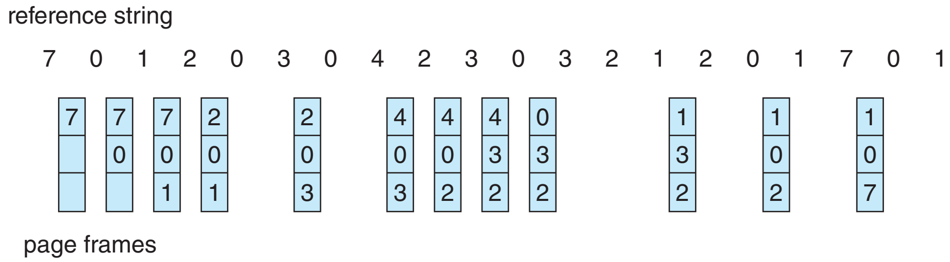


Figure 10.15 LRU page-replacement algorithm.

for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen.

The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is ***how*** to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

- **Counters**. In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

- **Stack**. Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom (Figure 10.16). Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Like optimal replacement, LRU replacement does not suffer from Belady's anomaly. Both belong to a class of page-replacement algorithms, called **stack algorithms**, that can never exhibit Belady's anomaly. A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for n frames is always a ***subset*** of the set of pages that would be in memory with n

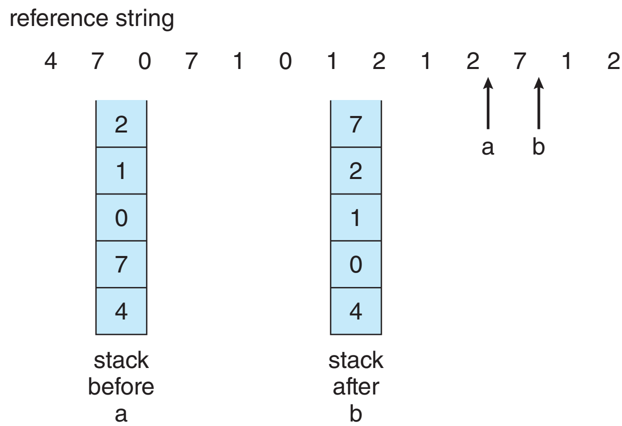


Figure 10.16 Use of a stack to record the most recent page references.

+ 1 frames. For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory.

Note that neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers. The updating of the clock fields or stack must be done for ***every*** memory reference. If we were to use an interrupt for every reference to allow software to update such data structures, it would slow every memory reference by a factor of at least ten, hence slowing every process by a factor of ten. Few systems could tolerate that level of overhead for memory management.

10.4.5 LRU-Approximation Page Replacement

Not many computer systems provide sufficient hardware support for true LRU page replacement. In fact, some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a **reference bit**. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the ***order*** of use. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

10.4.5.1 Additional-Reference-Bits Algorithm

We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, for example, then the page has not been used for eight time periods. A page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than one with a value of 01110111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value or use the FIFO method to choose among them.

The number of bits of history included in the shift register can be varied, of course, and is selected (depending on the hardware available) to make the updating as fast as possible. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the **second-chance page-replacement algorithm**.

10.4.5.2 Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

One way to implement the second-chance algorithm (sometimes referred to as the **clock** algorithm) is as a circular queue. A pointer (that is, a hand on the clock) indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 10.17). Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

10.4.5.3 Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify bit (described in Section 10.4.1) as an ordered pair. With these two bits, we have the following four possible classes:

1. (0, 0) neither recently used nor modified—best page to replace
2. (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean—probably will be used again soon

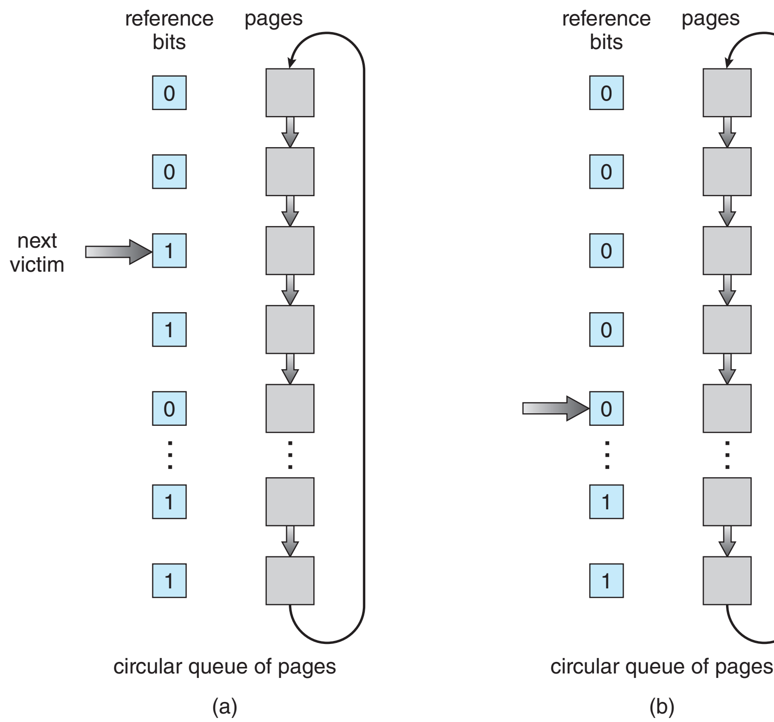


Figure 10.17 Second-chance (clock) page-replacement algorithm.

4. (1, 1) recently used and modified—probably will be used again soon, and the page will be need to be written out to secondary storage before it can be replaced

Each page is in one of these four classes. When page replacement is called for, we use the same scheme as in the clock algorithm; but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified in order to reduce the number of I/Os required.

10.4.6 Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

- The **least frequently used** (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

- The **most frequently used** (**MFU**) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

10.4.7 Page-Buffering Algorithms

Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to secondary storage. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.

Another modification is to keep a pool of free frames but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to secondary storage, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it.

Some versions of the UNIX system use this method in conjunction with the second-chance algorithm. It can be a useful augmentation to any page-replacement algorithm, to reduce the penalty incurred if the wrong victim page is selected. We describe these—and other—modifications in Section 10.5.3.

10.4.8 Applications and Page Replacement

In certain cases, applications accessing data through the operating system's virtual memory perform worse than if the operating system provided no buffering at all. A typical example is a database, which provides its own memory management and I/O buffering. Applications like this understand their memory use and storage use better than does an operating system that is implementing algorithms for general-purpose use. Furthermore, if the operating system is buffering I/O and the application is doing so as well, then twice the memory is being used for a set of I/O.

In another example, data warehouses frequently perform massive sequential storage reads, followed by computations and writes. The LRU algorithm would be removing old pages and preserving new ones, while the application would more likely be reading older pages than newer ones (as it starts its sequential reads again). Here, MFU would actually be more efficient than LRU.

Because of such problems, some operating systems give special programs the ability to use a secondary storage partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the **raw disk**, and I/O to this array is termed raw I/O. Raw I/O bypasses all the file-system services, such as file I/O demand paging, file locking, prefetching, space allocation, file names, and directories. Note that although certain applications are more efficient when implementing their own special-purpose storage services on a raw partition, most applications perform better when they use the regular file-system services.

10.5 Allocation of Frames

We turn next to the issue of allocation. How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?

Consider a simple case of a system with 128 frames. The operating system may take 35, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

There are many variations on this simple strategy. We can require that the operating system allocate all its buffer and table space from the free-frame list. When this space is not in use by the operating system, it can be used to support user paging. We can try to keep three free frames reserved on the free-frame list at all times. Thus, when a page fault occurs, there is a free frame available to page into. While the page swap is taking place, a replacement can be selected, which is then written to the storage device as the user process continues to execute. Other variants are also possible, but the basic strategy is clear: the user process is allocated any free frame.

10.5.1 Minimum Number of Frames

Our strategies for the allocation of frames are constrained in various ways. We cannot, for example, allocate more than the total number of available frames (unless there is page sharing). We must also allocate at least a minimum number of frames. Here, we look more closely at the latter requirement.

One reason for allocating at least a minimum number of frames involves performance. Obviously, as the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution. In addition, remember that, when a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently, we must have enough frames to hold all the different pages that any single instruction can reference.

For example, consider a machine in which all memory-reference instructions may reference only one memory address. In this case, we need at least one frame for the instruction and one frame for the memory reference. In addition, if one-level indirect addressing is allowed (for example, a `load` instruction on frame 16 can refer to an address on frame 0, which is an indirect reference to frame 23), then paging requires at least three frames per process. (Think about what might happen if a process had only two frames.)

The minimum number of frames is defined by the computer architecture. For example, if the move instruction for a given architecture includes more than one word for some addressing modes, the instruction itself may straddle two frames. In addition, if each of its two operands may be indirect references, a total of six frames are required. As another example, the move instruction for Intel 32- and 64-bit architectures allows data to move only from register to register and between registers and memory; it does not allow direct memory-to-memory movement, thereby limiting the required minimum number of frames for a process.

Whereas the minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory. In between, we are still left with significant choice in frame allocation.

10.5.2 Allocation Algorithms

The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames (ignoring frames needed by the operating system for the moment). For instance, if there are 93 frames and 5 processes, each process will get 18 frames. The 3 leftover frames can be used as a free-frame buffer pool. This scheme is called **equal allocation**.

An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.

To solve this problem, we can use **proportional allocation**, in which we allocate available memory to each process according to its size. Let the size of the virtual memory for process p_i be s_i, and define

$$S = \sum s_i.$$

Then, if the total number of available frames is m, we allocate a_i frames to process p_i, where a_i is approximately

$$a_i = s_i/S \times m.$$

Of course, we must adjust each a_i to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding m.

With proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since

$$10/137 \times 62 \approx 4 \text{ and}$$
$$127/137 \times 62 \approx 57.$$

In this way, both processes share the available frames according to their "needs," rather than equally.

In both equal and proportional allocation, of course, the allocation may vary according to the multiprogramming level. If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process. Conversely, if the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

Notice that, with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes. One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority.

10.5.3 Global versus Local Allocation

Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement**. Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

For example, consider an allocation scheme wherein we allow high-priority processes to select frames from low-priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process. Whereas with a local replacement strategy, the number of frames allocated to a process does not change, with global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes do not choose ***its*** frames for replacement).

One problem with a global replacement algorithm is that the set of pages in memory for a process depends not only on the paging behavior of that process, but also on the paging behavior of other processes. Therefore, the same process may perform quite differently (for example, taking 0.5 seconds for one execution and 4.3 seconds for the next execution) because of totally external circumstances. Such is not the case with a local replacement algorithm. Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. Local replacement might hinder a process, however, by not making available to it other, less used pages of memory. Thus, global replacement generally results in greater system throughput. It is therefore the more commonly used method.

MAJOR AND MINOR PAGE FAULTS

As described in Section 10.2.1, a page fault occurs when a page does not have a valid mapping in the address space of a process. Operating systems generally distinguish between two types of page faults: **major** and **minor** faults. (Windows refers to major and minor faults as **hard** and **soft** faults, respectively.) A major page fault occurs when a page is referenced and the page is not in memory. Servicing a major page fault requires reading the desired page from the backing store into a free frame and updating the page table. Demand paging typically generates an initially high rate of major page faults.

Minor page faults occur when a process does not have a logical mapping to a page, yet that page is in memory. Minor faults can occur for one of two reasons. First, a process may reference a shared library that is in memory, but the process does not have a mapping to it in its page table. In this instance, it is only necessary to update the page table to refer to the existing page in memory. A second cause of minor faults occurs when a page is reclaimed from a process and placed on the free-frame list, but the page has not yet been zeroed out and allocated to another process. When this kind of fault occurs, the frame is removed from the free-frame list and reassigned to the process. As might be expected, resolving a minor page fault is typically much less time consuming than resolving a major page fault.

You can observe the number of major and minor page faults in a Linux system using the command `ps -eo min_flt,maj_flt,cmd`, which outputs the number of minor and major page faults, as well as the command that launched the process. An example output of this `ps` command appears below:

```
MINFL     MAJFL     CMD
186509    32        /usr/lib/systemd/systemd-logind
76822     9         /usr/sbin/sshd -D
1937      0         vim 10.tex
699       14        /sbin/auditd -n
```

Here, it is interesting to note that, for most commands, the number of major page faults is generally quite low, whereas the number of minor faults is much higher. This indicates that Linux processes likely take significant advantage of shared libraries as, once a library is loaded in memory, subsequent page faults are only minor faults.

Next, we focus on one possible strategy that we can use to implement a global page-replacement policy. With this approach, we satisfy all memory requests from the free-frame list, but rather than waiting for the list to drop to zero before we begin selecting pages for replacement, we trigger page replacement when the list falls below a certain threshold. This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

Such a strategy is depicted in Figure 10.18. The strategy's purpose is to keep the amount of free memory above a minimum threshold. When it drops

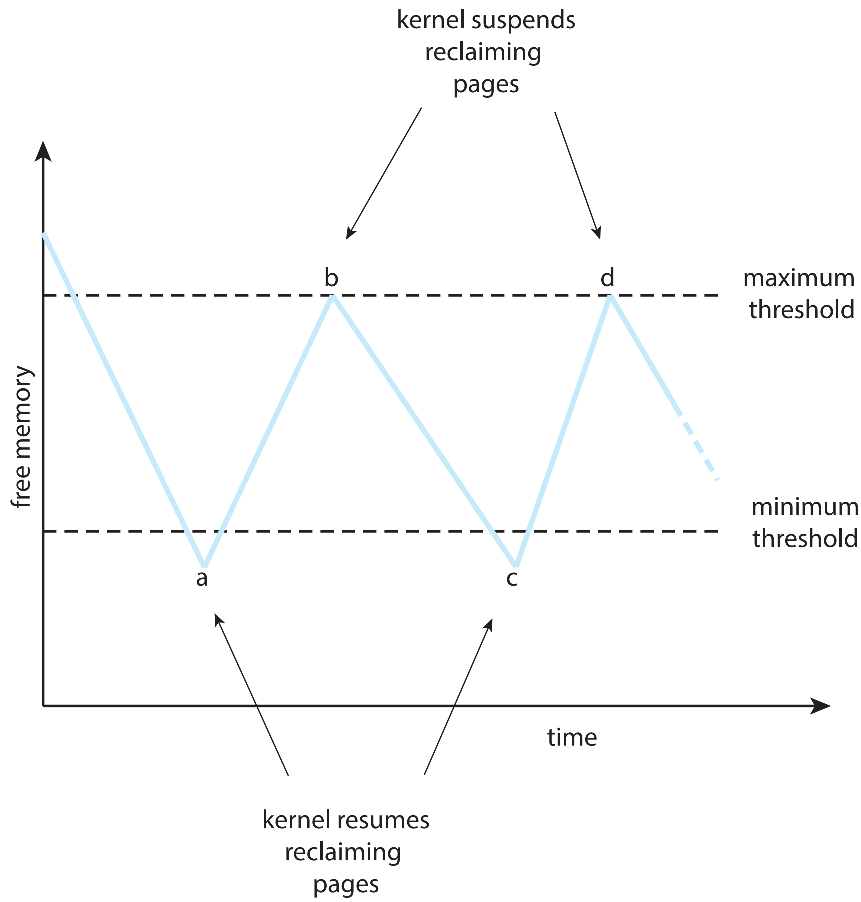


Figure 10.18 Reclaiming pages.

below this threshold, a kernel routine is triggered that begins reclaiming pages from all processes in the system (typically excluding the kernel). Such kernel routines are often known as **reapers**, and they may apply any of the page-replacement algorithms covered in Section 10.4. When the amount of free memory reaches the maximum threshold, the reaper routine is suspended, only to resume once free memory again falls below the minimum threshold.

In Figure 10.18, we see that at point a the amount of free memory drops below the minimum threshold, and the kernel begins reclaiming pages and adding them to the free-frame list. It continues until the maximum threshold is reached (point b). Over time, there are additional requests for memory, and at point c the amount of free memory again falls below the minimum threshold. Page reclamation resumes, only to be suspended when the amount of free memory reaches the maximum threshold (point d). This process continues as long as the system is running.

As mentioned above, the kernel reaper routine may adopt any page-replacement algorithm, but typically it uses some form of LRU approximation. Consider what may happen, though, if the reaper routine is unable to maintain the list of free frames below the minimum threshold. Under these circumstances, the reaper routine may begin to reclaim pages more aggressively. For example, perhaps it will suspend the second-chance algorithm and use pure FIFO. Another, more extreme, example occurs in Linux; when the amount of free memory falls to ***very*** low levels, a routine known as the **out-of-memory (OOM) killer** selects a process to terminate, thereby freeing its memory. How does Linux determine which process to terminate? Each process has what is known as an OOM score, with a higher score increasing the likelihood that the process could be terminated by the OOM killer routine. OOM scores are calculated according to the percentage of memory a process is using—the higher the percentage, the higher the OOM score. (OOM scores can be viewed in the `/proc` file system, where the score for a process with `pid` 2500 can be viewed as `/proc/2500/oom_score`.)

In general, not only can reaper routines vary how aggressively they reclaim memory, but the values of the minimum and maximum thresholds can be varied as well. These values can be set to default values, but some systems may allow a system administrator to configure them based on the amount of physical memory in the system.

10.5.4 Non-Uniform Memory Access

Thus far in our coverage of virtual memory, we have assumed that all main memory is created equal—or at least that it is accessed equally. On **non-uniform memory access (NUMA)** systems with multiple CPUs (Section 1.3.2), that is not the case. On these systems, a given CPU can access some sections of main memory faster than it can access others. These performance differences are caused by how CPUs and memory are interconnected in the system. Such a system is made up of multiple CPUs, each with its own local memory (Figure 10.19). The CPUs are organized using a shared system interconnect, and as you might expect, a CPU can access its local memory faster than memory local to another CPU. NUMA systems are without exception slower than systems in which all accesses to main memory are treated equally. However, as described in Section 1.3.2, NUMA systems can accommodate more CPUs and therefore achieve greater levels of throughput and parallelism.

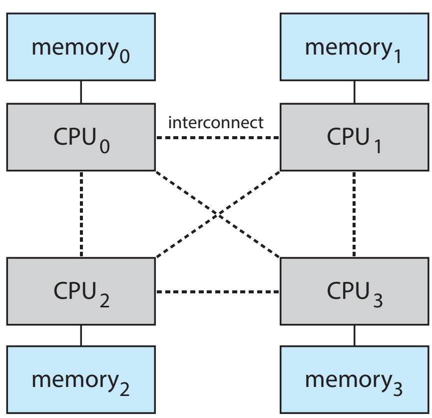


Figure 10.19 NUMA multiprocessing architecture.

Managing which page frames are stored at which locations can significantly affect performance in NUMA systems. If we treat memory as uniform in such a system, CPUs may wait significantly longer for memory access than if we modify memory allocation algorithms to take NUMA into account. We described some of these modifications in Section 5.5.4. Their goal is to have memory frames allocated "as close as possible" to the CPU on which the process is running. (The definition of ***close*** is "with minimum latency," which typically means on the same system board as the CPU). Thus, when a process incurs a page fault, a NUMA-aware virtual memory system will allocate that process a frame as close as possible to the CPU on which the process is running.

To take NUMA into account, the scheduler must track the last CPU on which each process ran. If the scheduler tries to schedule each process onto its previous CPU, and the virtual memory system tries to allocate frames for the process close to the CPU on which it is being scheduled, then improved cache hits and decreased memory access times will result.

The picture is more complicated once threads are added. For example, a process with many running threads may end up with those threads scheduled on many different system boards. How should the memory be allocated in this case?

As we discussed in Section 5.7.1, Linux manages this situation by having the kernel identify a hierarchy of scheduling domains. The Linux CFS scheduler does not allow threads to migrate across different domains and thus incur memory access penalties. Linux also has a separate free-frame list for each NUMA node, thereby ensuring that a thread will be allocated memory from the node on which it is running. Solaris solves the problem similarly by creating **lgroups** (for "locality groups") in the kernel. Each lgroup gathers together CPUs and memory, and each CPU in that group can access any memory in the group within a defined latency interval. In addition, there is a hierarchy of lgroups based on the amount of latency between the groups, similar to the hierarchy of scheduling domains in Linux. Solaris tries to schedule all threads of a process and allocate all memory of a process within an lgroup. If that is not possible, it picks nearby lgroups for the rest of the resources needed. This practice minimizes overall memory latency and maximizes CPU cache hit rates.

10.6 Thrashing

Consider what occurs if a process does not have "enough" frames—that is, it does not have the minimum number of frames it needs to support pages in the working set. The process will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing. As you might expect, thrashing results in severe performance problems.

10.6.1 Cause of Thrashing

Consider the following scenario, which is based on the actual behavior of early paging systems. The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization and ***increases*** the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The page-fault rate increases tremendously. As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time paging.

This phenomenon is illustrated in Figure 10.20, in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must ***decrease*** the degree of multiprogramming.

We can limit the effects of thrashing by using a **local replacement algorithm** (or **priority replacement algorithm**). As mentioned earlier, local replacement requires that each process select from only its own set of allocated frames. Thus, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. However, the problem is not entirely solved. If processes are thrashing, they will be in the queue for the paging device most of the time. The average service time for a page fault will increase

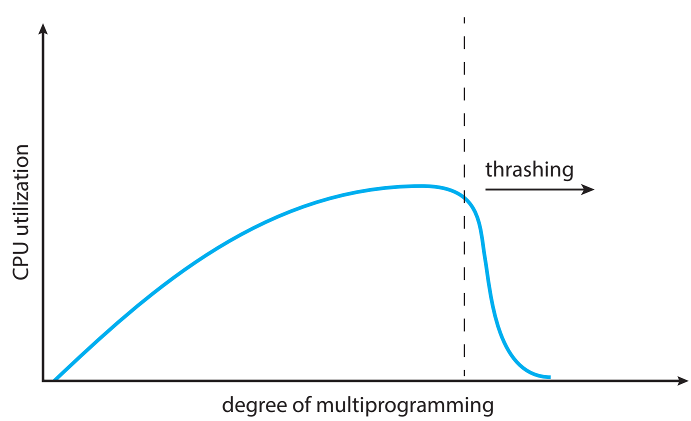


Figure 10.20 Thrashing.

because of the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

To prevent thrashing, we must provide a process with as many frames as it needs. But how do we know how many frames it "needs"? One strategy starts by looking at how many frames a process is actually using. This approach defines the **locality model** of process execution.

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A running program is generally composed of several different localities, which may overlap. For example, when a function is called, it defines a new locality. In this

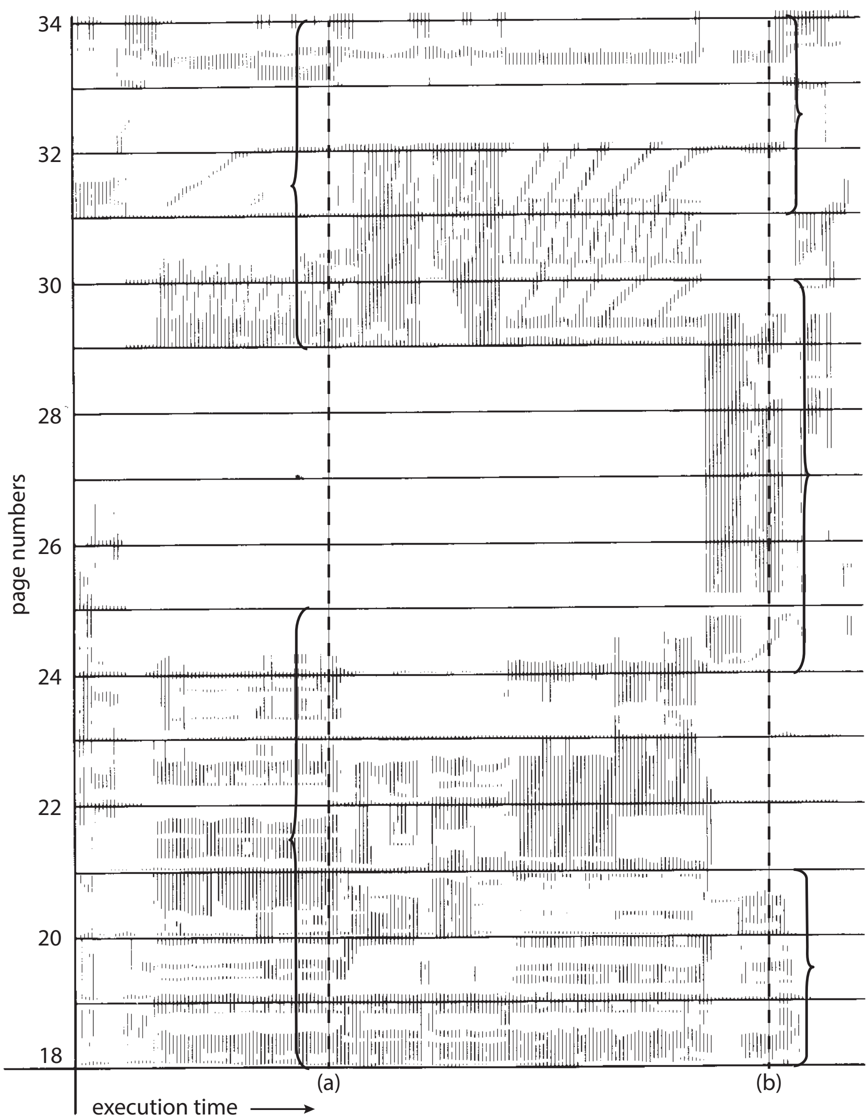


Figure 10.21 Locality in a memory-reference pattern.

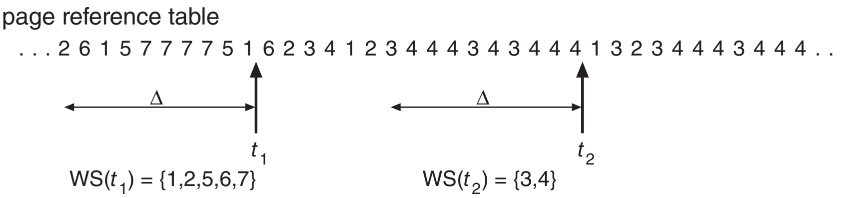


Figure 10.22 Working-set model.

locality, memory references are made to the instructions of the function call, its local variables, and a subset of the global variables. When we exit the function, the process leaves this locality, since the local variables and instructions of the function are no longer in active use. We may return to this locality later.

Figure 10.21 illustrates the concept of locality and how a process's locality changes over time. At time (*a*), the locality is the set of pages $\{18, 19, 20, 21, 22, 23, 24, 29, 30, 33\}$. At time (*b*), the locality changes to $\{18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33\}$. Notice the overlap, as some pages (for example, 18, 19, and 20) are part of both localities.

Thus, we see that localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure. Note that the locality model is the unstated principle behind the caching discussions so far in this book. If accesses to any types of data were random rather than patterned, caching would be useless.

Suppose we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities. If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

10.6.2 Working-Set Model

The **working-set model** is based on the assumption of locality. This model uses a parameter, Δ, to define the **working-set window**. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is the **working set** (Figure 10.22). If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. Thus, the working set is an approximation of the program's locality.

For example, given the sequence of memory references shown in Figure 10.22, if $\Delta = 10$ memory references, then the working set at time t_1 is $\{1, 2, 5, 6, 7\}$. By time t_2, the working set has changed to $\{3, 4\}$.

The accuracy of the working set depends on the selection of Δ. If Δ is too small, it will not encompass the entire locality; if Δ is too large, it may overlap several localities. In the extreme, if Δ is infinite, the working set is the set of pages touched during the process execution.

The most important property of the working set, then, is its size. If we compute the working-set size, WSS_i, for each process in the system, we can then consider that

$$D = \sum WSS_i,$$

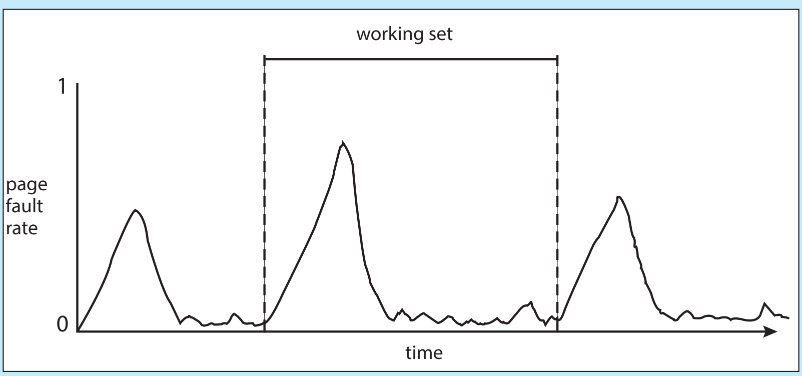
where D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs WSS_i frames. If the total demand is greater than the total number of available frames ($D > m$), thrashing will occur, because some processes will not have enough frames.

Once Δ has been selected, use of the working-set model is simple. The operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated. If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process's pages are written out (swapped), and its frames are reallocated to other processes. The suspended process can be restarted later.

This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization. The difficulty with the working-set model is keeping track of the working set. The

WORKING SETS AND PAGE-FAULT RATES

There is a direct relationship between the working set of a process and its page-fault rate. Typically, as shown in Figure 10.22, the working set of a process changes over time as references to data and code sections move from one locality to another. Assuming there is sufficient memory to store the working set of a process (that is, the process is not thrashing), the page-fault rate of the process will transition between peaks and valleys over time. This general behavior is shown below:



A peak in the page-fault rate occurs when we begin demand-paging a new locality. However, once the working set of this new locality is in memory, the page-fault rate falls. When the process moves to a new working set, the page-fault rate rises toward a peak once again, returning to a lower rate once the new working set is loaded into memory. The span of time between the start of one peak and the start of the next peak represents the transition from one working set to another.

working-set window is a moving window. At each memory reference, a new reference appears at one end, and the oldest reference drops off the other end. A page is in the working set if it is referenced anywhere in the working-set window.

We can approximate the working-set model with a fixed-interval timer interrupt and a reference bit. For example, assume that Δ equals 10,000 references and that we can cause a timer interrupt every 5,000 references. When we get a timer interrupt, we copy and clear the reference-bit values for each page. Thus, if a page fault occurs, we can examine the current reference bit and two in-memory bits to determine whether a page was used within the last 10,000 to 15,000 references. If it was used, at least one of these bits will be on. If it has not been used, these bits will be off. Pages with at least one bit on will be considered to be in the working set.

Note that this arrangement is not entirely accurate, because we cannot tell where, within an interval of 5,000, a reference occurred. We can reduce the uncertainty by increasing the number of history bits and the frequency of interrupts (for example, 10 bits and interrupts every 1,000 references). However, the cost to service these more frequent interrupts will be correspondingly higher.

10.6.3 Page-Fault Frequency

The working-set model is successful, and knowledge of the working set can be useful for prepaging (Section 10.9.1), but it seems a clumsy way to control thrashing. A strategy that uses the **page-fault frequency** (**PFF**) takes a more direct approach.

The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate (Figure 10.23). If the actual page-fault rate exceeds the upper limit, we allocate the process

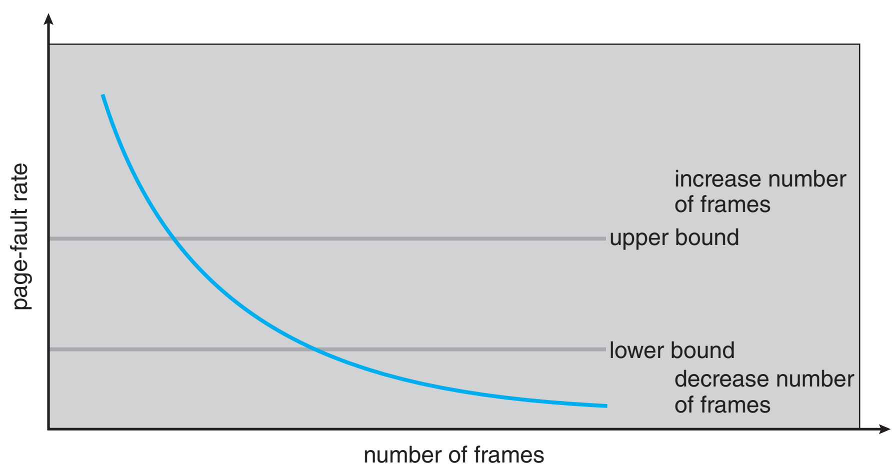


Figure 10.23 Page-fault frequency.

another frame. If the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

As with the working-set strategy, we may have to swap out a process. If the page-fault rate increases and no free frames are available, we must select some process and swap it out to backing store. The freed frames are then distributed to processes with high page-fault rates.

10.6.4 Current Practice

Practically speaking, thrashing and the resulting swapping have a disagreeably high impact on performance. The current best practice in implementing a computer system is to include enough physical memory, whenever possible, to avoid thrashing and swapping. From smartphones through large servers, providing enough memory to keep all working sets in memory concurrently, except under extreme conditions, provides the best user experience.

10.7 Memory Compression

An alternative to paging is **memory compression**. Here, rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.

In Figure 10.24, the free-frame list contains six frames. Assume that this number of free frames falls below a certain threshold that triggers page replacement. The replacement algorithm (say, an LRUapproximation algorithm) selects four frames—15, 3, 35, and 26—to place on the free-frame list. It first places these frames on a modified-frame list. Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list. An alternative strategy is to compress a number of frames—say, three—and store their compressed versions in a single page frame.

In Figure 10.25, frame 7 is removed from the free-frame list. Frames 15, 3, and 35 are compressed and stored in frame 7, which is then stored in the list of compressed frames. The frames 15, 3, and 35 can now be moved to the free-frame list. If one of the three compressed frames is later referenced, a page fault occurs, and the compressed frame is decompressed, restoring the three pages 15, 3, and 35 in memory.

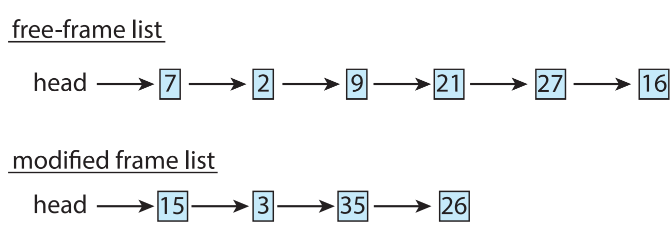


Figure 10.24 Free-frame list before compression.

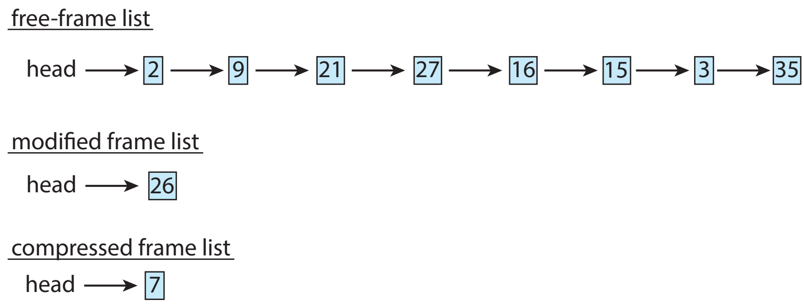


Figure 10.25 Free-frame list after compression

As we have noted, mobile systems generally do not support either standard swapping or swapping pages. Thus, memory compression is an integral part of the memory-management strategy for most mobile operating systems, including Android and iOS. In addition, both Windows 10 and macOS support memory compression. For Windows 10, Microsoft developed the **Universal Windows Platform** (**UWP**) architecture, which provides a common app platform for devices that run Windows 10, including mobile devices. UWP apps running on mobile devices are candidates for memory compression. macOS first supported memory compression with Version 10.9 of the operating system, first compressing LRU pages when free memory is short and then paging if that doesn't solve the problem. Performance tests indicate that memory compression is faster than paging even to SSD secondary storage on laptop and desktop macOS systems.

Although memory compression does require allocating free frames to hold the compressed pages, a significant memory saving can be realized, depending on the reductions achieved by the compression algorithm. (In the example above, the three frames were reduced to one-third of their original size.) As with any form of data compression, there is contention between the speed of the compression algorithm and the amount of reduction that can be achieved (known as the **compression ratio**). In general, higher compression ratios (greater reductions) can be achieved by slower, more computationally expensive algorithms. Most algorithms in use today balance these two factors, achieving relatively high compression ratios using fast algorithms. In addition, compression algorithms have improved by taking advantage of multiple computing cores and performing compression in parallel. For example, Microsoft's Xpress and Apple's WKdm compression algorithms are considered fast, and they report compressing pages to 30 to 50 percent of their original size.

10.8 Allocating Kernel Memory

When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel. This list is typically populated using a page-replacement algorithm such as those discussed in Section 10.4 and most likely contains free pages scattered throughout physical memory, as explained earlier. Remember, too, that if a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.

Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.

2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface—and consequently may require memory residing in physically contiguous pages.

In the following sections, we examine two strategies for managing free memory that is assigned to kernel processes: the "buddy system" and slab allocation.

10.8.1 Buddy System

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16-KB segment.

Let's consider a simple example. Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. The segment is initially divided into two **buddies**—which we will call A_L and A_R—each 128 KB in size. One of these buddies is further divided into two 64-KB buddies—B_L and B_R. However, the next-highest power of 2 from 21 KB is 32 KB so either B_L or B_R is again divided into two 32-KB buddies, C_L and C_R. One of these buddies is used to satisfy the 21-KB request. This scheme is illustrated in Figure 10.26, where C_L is the segment allocated to the 21-KB request.

An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as **coalescing**. In Figure 10.26, for example, when the kernel releases the C_L unit it was allocated, the system can coalesce C_L and C_R into a 64-KB segment. This segment, B_L, can in turn be coalesced with its buddy B_R to form a 128-KB segment. Ultimately, we can end up with the original 256-KB segment.

The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments. For example, a 33-KB request can only be satisfied with a 64-KB segment. In fact, we cannot guarantee that less than 50 percent of the allocated unit will be wasted due to internal fragmentation. In the following section, we explore a memory allocation scheme where no space is lost due to fragmentation.

10.8.2 Slab Allocation

A second strategy for allocating kernel memory is known as **slab allocation**. A **slab** is made up of one or more physically contiguous pages. A **cache** consists

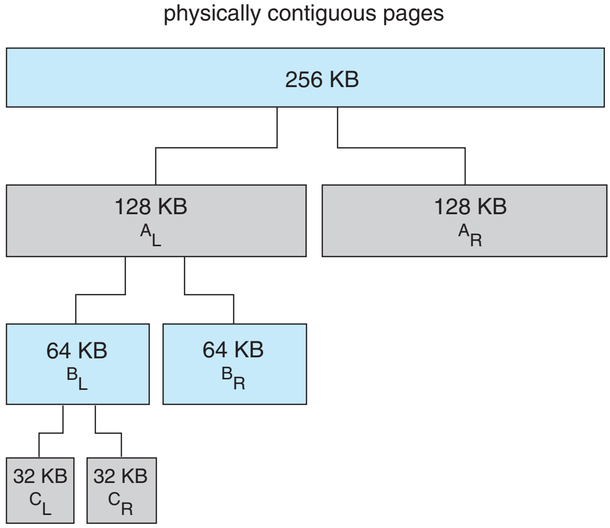


Figure 10.26 Buddy system allocation.

of one or more slabs. There is a single cache for each unique kernel data structure—for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth. Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects, and so forth. The relationship among slabs, caches, and objects is shown in Figure 10.27. The figure shows two kernel objects 3 KB in size and three objects 7 KB in size, each stored in a separate cache.

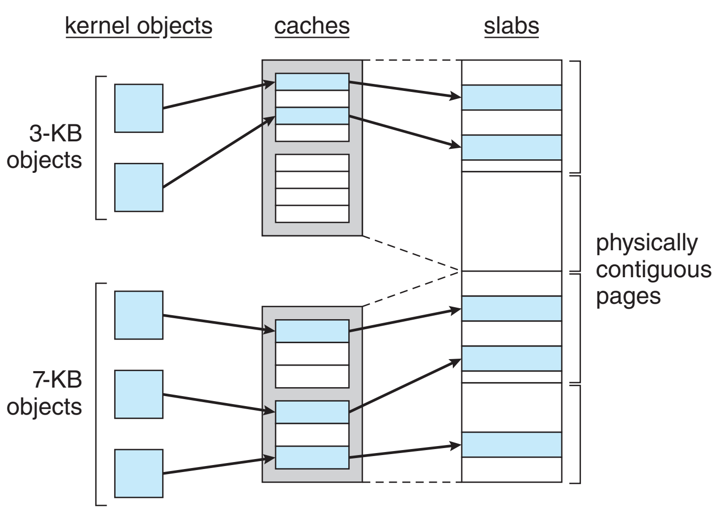


Figure 10.27 Slab allocation.

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects—which are initially marked as `free`—are allocated to the cache. The number of objects in the cache depends on the size of the associated slab. For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as `used`.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type `struct task_struct`, which requires approximately 1.7 KB of memory. When the Linux kernel creates a new task, it requests the necessary memory for the `struct task_struct` object from its cache. The cache will fulfill the request using a `struct task_struct` object that has already been allocated in a slab and is marked as free.

In Linux, a slab may be in one of three possible states:

1. **Full**. All objects in the slab are marked as used.
2. **Empty**. All objects in the slab are marked as free.
3. **Partial**. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

The slab allocator provides two main benefits:

1. No memory is wasted due to fragmentation. Fragmentation is not an issue because each unique kernel data structure has an associated cache, and each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented. Thus, when the kernel requests memory for an object, the slab allocator returns the exact amount of memory required to represent the object.
2. Memory requests can be satisfied quickly. The slab allocation scheme is thus particularly effective for managing memory when objects are frequently allocated and deallocated, as is often the case with requests from the kernel. The act of allocating—and releasing—memory can be a time-consuming process. However, objects are created in advance and thus can be quickly allocated from the cache. Furthermore, when the kernel has finished with an object and releases it, it is marked as free and returned to its cache, thus making it immediately available for subsequent requests from the kernel.

The slab allocator first appeared in the Solaris 2.4 kernel. Because of its general-purpose nature, this allocator is now also used for certain user-mode memory requests in Solaris. Linux originally used the buddy system; however, beginning with Version 2.2, the Linux kernel adopted the slab allocator. Linux refers to its slab implementation as SLAB. Recent distributions of Linux include two other kernel memory allocators—the SLOB and SLUB allocators.

The SLOB allocator is designed for systems with a limited amount of memory, such as embedded systems. SLOB (which stands for "simple list of blocks") maintains three lists of objects: ***small*** (for objects less than 256 bytes), ***medium*** (for objects less than 1,024 bytes), and ***large*** (for all other objects less than the size of a page). Memory requests are allocated from an object on the appropriate list using a first-fit policy.

Beginning with Version 2.6.24, the SLUB allocator replaced SLAB as the default allocator for the Linux kernel. SLUB reduced much of the overhead required by the SLAB allocator. For instance, whereas SLAB stores certain metadata with each slab, SLUB stores these data in the `page` structure the Linux kernel uses for each page. Additionally, SLUB does not include the per-CPU queues that the SLAB allocator maintains for objects in each cache. For systems with a large number of processors, the amount of memory allocated to these queues is significant. Thus, SLUB provides better performance as the number of processors on a system increases.

10.9 Other Considerations

The major decisions that we make for a paging system are the selections of a replacement algorithm and an allocation policy, which we discussed earlier in this chapter. There are many other considerations as well, and we discuss several of them here.

10.9.1 Prepaging

An obvious property of pure demand paging is the large number of page faults that occur when a process is started. This situation results from trying to get the initial locality into memory. **Prepaging** is an attempt to prevent this high level of initial paging. The strategy is to bring some—or all—of the pages that will be needed into memory at one time.

In a system using the working-set model, for example, we could keep with each process a list of the pages in its working set. If we must suspend a process (due to a lack of free frames), we remember the working set for that process. When the process is to be resumed (because I/O has finished or enough free frames have become available), we automatically bring back into memory its entire working set before restarting the process.

Prepaging may offer an advantage in some cases. The question is simply whether the cost of using prepaging is less than the cost of servicing the corresponding page faults. It may well be the case that many of the pages brought back into memory by prepaging will not be used.

Assume that s pages are prepaged and a fraction α of these s pages is actually used ($0 \leq \alpha \leq 1$). The question is whether the cost of the $s * \alpha$ saved page faults is greater or less than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages. If α is close to 0, prepaging loses; if α is close to 1, prepaging wins.

Note also that prepaging an executable program may be difficult, as it may be unclear exactly what pages should be brought in. Prepaging a file may be more predictable, since files are often accessed sequentially. The Linux readahead() system call prefetches the contents of a file into memory so that subsequent accesses to the file will take place in main memory.

10.9.2 Page Size

The designers of an operating system for an existing machine seldom have a choice concerning the page size. However, when new machines are being designed, a decision regarding the best page size must be made. As you might expect, there is no single best page size. Rather, there is a set of factors that support various sizes. Page sizes are invariably powers of 2, generally ranging from 4,096 (2^{12}) to 4,194,304 (2^{22}) bytes.

How do we select a page size? One concern is the size of the page table. For a given virtual memory space, decreasing the page size increases the number of pages and hence the size of the page table. For a virtual memory of 4 MB (2^{22}), for example, there would be 4,096 pages of 1,024 bytes but only 512 pages of 8,192 bytes. Because each active process must have its own copy of the page table, a large page size is desirable.

Memory is better utilized with smaller pages, however. If a process is allocated memory starting at location 00000 and continuing until it has as much as it needs, it probably will not end exactly on a page boundary. Thus, a part of the final page must be allocated (because pages are the units of allocation) but will be unused (creating internal fragmentation). Assuming independence of process size and page size, we can expect that, on the average, half of the final page of each process will be wasted. This loss is only 256 bytes for a page of 512 bytes but is 4,096 bytes for a page of 8,192 bytes. To minimize internal fragmentation, then, we need a small page size.

Another problem is the time required to read or write a page. As you will see in Section 11.1, when the storage device is an HDD, I/O time is composed of seek, latency, and transfer times. Transfer time is proportional to the amount transferred (that is, the page size)—a fact that would seem to argue for a small page size. However, latency and seek time normally dwarf transfer time. At a transfer rate of 50 MB per second, it takes only 0.01 milliseconds to transfer 512 bytes. Latency time, though, is perhaps 3 milliseconds, and seek time 5 milliseconds. Of the total I/O time (8.01 milliseconds), therefore, only about 0.1 percent is attributable to the actual transfer. Doubling the page size increases I/O time to only 8.02 milliseconds. It takes 8.02 milliseconds to read a single page of 1,024 bytes but 16.02 milliseconds to read the same amount as two pages of 512 bytes each. Thus, a desire to minimize I/O time argues for a larger page size.

With a smaller page size, though, total I/O should be reduced, since locality will be improved. A smaller page size allows each page to match program locality more accurately. For example, consider a process 200 KB in size, of which only half (100 KB) is actually used in an execution. If we have only one large page, we must bring in the entire page, a total of 200 KB transferred and allocated. If instead we had pages of only 1 byte, then we could bring in only the 100 KB that are actually used, resulting in only 100 KB transferred and allocated. With a smaller page size, then, we have better **resolution**, allowing us to isolate only the memory that is actually needed. With a larger page size, we must allocate and transfer not only what is needed but also anything else that happens to be in the page, whether it is needed or not. Thus, a smaller page size should result in less I/O and less total allocated memory.

But did you notice that with a page size of 1 byte, we would have a page fault for ***each*** byte? A process of 200 KB that used only half of that memory would generate only one page fault with a page size of 200 KB but 102,400 page faults with a page size of 1 byte. Each page fault generates the large amount of overhead needed for processing the interrupt, saving registers, replacing a page, queuing for the paging device, and updating tables. To minimize the number of page faults, we need to have a large page size.

Other factors must be considered as well (such as the relationship between page size and sector size on the paging device). The problem has no best answer. As we have seen, some factors (internal fragmentation, locality) argue for a small page size, whereas others (table size, I/O time) argue for a large page size. Nevertheless, the historical trend is toward larger page sizes, even for mobile systems. Indeed, the first edition of *Operating System Concepts* (1983) used 4,096 bytes as the upper bound on page sizes, and this value was the most common page size in 1990. Modern systems may now use much larger page sizes, as you will see in the following section.

10.9.3 TLB Reach

In Chapter 9, we introduced the **hit ratio** of the TLB. Recall that the hit ratio for the TLB refers to the percentage of virtual address translations that are resolved in the TLB rather than the page table. Clearly, the hit ratio is related to the number of entries in the TLB, and the way to increase the hit ratio is by increasing the number of entries. This, however, does not come cheaply, as the associative memory used to construct the TLB is both expensive and power hungry.

Related to the hit ratio is a similar metric: the **TLB reach**. The TLB reach refers to the amount of memory accessible from the TLB and is simply the number of entries multiplied by the page size. Ideally, the working set for a process is stored in the TLB. If it is not, the process will spend a considerable amount of time resolving memory references in the page table rather than the TLB. If we double the number of entries in the TLB, we double the TLB reach. However, for some memory-intensive applications, this may still prove insufficient for storing the working set.

Another approach for increasing the TLB reach is to either increase the size of the page or provide multiple page sizes. If we increase the page size—say, from 4 KB to 16 KB—we quadruple the TLB reach. However, this may lead to an increase in fragmentation for some applications that do not require such a large page size. Alternatively, most architectures provide support for more than one page size, and an operating system can be configured to take advantage of this support. For example, the default page size on Linux systems is 4 KB; however, Linux also provides **huge pages**, a feature that designates a region of physical memory where larger pages (for example, 2 MB) may be used.

Recall from Section 9.7 that the ARMv8 architecture provides support for pages and regions of different sizes. Additionally, each TLB entry in the ARMv8 contains a **contiguous bit**. If this bit is set for a particular TLB entry, that entry maps contiguous (adjacent) blocks of memory. Three possible arrangements of contiguous blocks can be mapped in a single TLB entry, thereby increasing the TLB reach:

1. 64-KB TLB entry comprising 16×4 KB adjacent blocks.
2. 1-GB TLB entry comprising 32×32 MB adjacent blocks.
3. 2-MB TLB entry comprising either 32×64 KB adjacent blocks, or 128×16 KB adjacent blocks.

Providing support for multiple page sizes may require the operating system—rather than hardware—to manage the TLB. For example, one of the fields in a TLB entry must indicate the size of the page frame corresponding to the entry—or, in the case of ARM architectures, must indicate that the entry refers to a contiguous block of memory. Managing the TLB in software and not hardware comes at a cost in performance. However, the increased hit ratio and TLB reach offset the performance costs.

10.9.4 Inverted Page Tables

Section 9.4.3 introduced the concept of the inverted page table. The purpose of this form of page management is to reduce the amount of physical memory needed to track virtual-to-physical address translations. We accomplish this savings by creating a table that has one entry per page of physical memory, indexed by the pair <process-id, page-number>.

Because they keep information about which virtual memory page is stored in each physical frame, inverted page tables reduce the amount of physical memory needed to store this information. However, the inverted page table no longer contains complete information about the logical address space of a process, and that information is required if a referenced page is not currently in memory. Demand paging requires this information to process page faults. For the information to be available, an external page table (one per process) must be kept. Each such table looks like the traditional per-process page table and contains information on where each virtual page is located.

But do external page tables negate the utility of inverted page tables? Since these tables are referenced only when a page fault occurs, they do not need to be available quickly. Instead, they are themselves paged in and out of memory as necessary. Unfortunately, a page fault may now cause the virtual memory manager to generate another page fault as it pages in the external page table it needs to locate the virtual page on the backing store. This special case requires careful handling in the kernel and a delay in the page-lookup processing.

10.9.5 Program Structure

Demand paging is designed to be transparent to the user program. In many cases, the user is completely unaware of the paged nature of memory. In other cases, however, system performance can be improved if the user (or compiler) has an awareness of the underlying demand paging.

Let's look at a contrived but informative example. Assume that pages are 128 words in size. Consider a C program whose function is to initialize to 0 each element of a 128-by-128 array. The following code is typical:

```
int i, j;
int[128][128] data;

for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
       data[i][j] = 0;
```

Notice that the array is stored row major; that is, the array is stored `data[0][0]`, `data[0][1]`, ···, `data[0][127]`, `data[1][0]`, `data[1][1]`, ···, `data[127][127]`. For pages of 128 words, each row takes one page. Thus, the preceding code zeros one word in each page, then another word in each page, and so on. If the operating system allocates fewer than 128 frames to the entire program, then its execution will result in $128 \times 128 = 16{,}384$ page faults.

In contrast, suppose we change the code to

```
int i, j;
int[128][128] data;

for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
       data[i][j] = 0;
```

This code zeros all the words on one page before starting the next page, reducing the number of page faults to 128.

Careful selection of data structures and programming structures can increase locality and hence lower the page-fault rate and the number of pages in the working set. For example, a stack has good locality, since access is always made to the top. A hash table, in contrast, is designed to scatter references, producing bad locality. Of course, locality of reference is just one measure of the efficiency of the use of a data structure. Other heavily weighted factors include search speed, total number of memory references, and total number of pages touched.

At a later stage, the compiler and loader can have a significant effect on paging. Separating code and data and generating reentrant code means that code pages can be read-only and hence will never be modified. Clean pages do not have to be paged out to be replaced. The loader can avoid placing routines across page boundaries, keeping each routine completely in one page. Routines that call each other many times can be packed into the same page. This packaging is a variant of the bin-packing problem of operations research: try to pack the variable-sized load segments into the fixed-sized pages so that interpage references are minimized. Such an approach is particularly useful for large page sizes.

10.9.6 I/O Interlock and Page Locking

When demand paging is used, we sometimes need to allow some of the pages to be **locked** in memory. One such situation occurs when I/O is done to or from user (virtual) memory. I/O is often implemented by a separate I/O processor. For example, a controller for a USB storage device is generally given the number of bytes to transfer and a memory address for the buffer (Figure 10.28). When the transfer is complete, the CPU is interrupted.

We must be sure the following sequence of events does not occur: A process issues an I/O request and is put in a queue for that I/O device. Meanwhile, the CPU is given to other processes. These processes cause page faults, and one of them, using a global replacement algorithm, replaces the page containing the memory buffer for the waiting process. The pages are paged out. Some time later, when the I/O request advances to the head of the device queue, the I/O occurs to the specified address. However, this frame is now being used for a different page belonging to another process.

There are two common solutions to this problem. One solution is never to execute I/O to user memory. Instead, data are always copied between system memory and user memory. I/O takes place only between system memory and the I/O device. Thus, to write a block on tape, we first copy the block to system memory and then write it to tape. This extra copying may result in unacceptably high overhead.

Another solution is to allow pages to be locked into memory. Here, a lock bit is associated with every frame. If the frame is locked, it cannot be selected for replacement. Under this approach, to write a block to disk, we lock into memory the pages containing the block. The system can then continue as usual. Locked pages cannot be replaced. When the I/O is complete, the pages are unlocked.

Lock bits are used in various situations. Frequently, some or all of the operating-system kernel is locked into memory. Many operating systems cannot tolerate a page fault caused by the kernel or by a specific kernel module, including the one performing memory management. User processes may also need to lock pages into memory. A database process may want to manage a chunk of memory, for example, moving blocks between secondary storage and

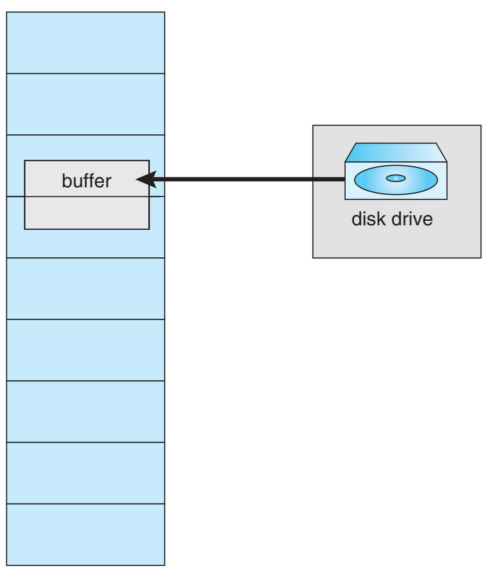


Figure 10.28 The reason why frames used for I/O must be in memory.

memory itself because it has the best knowledge of how it is going to use its data. Such **pinning** of pages in memory is fairly common, and most operating systems have a system call allowing an application to request that a region of its logical address space be pinned. Note that this feature could be abused and could cause stress on the memory-management algorithms. Therefore, an application frequently requires special privileges to make such a request.

Another use for a lock bit involves normal page replacement. Consider the following sequence of events: A low-priority process faults. Selecting a replacement frame, the paging system reads the necessary page into memory. Ready to continue, the low-priority process enters the ready queue and waits for the CPU. Since it is a low-priority process, it may not be selected by the CPU scheduler for a time. While the low-priority process waits, a high-priority process faults. Looking for a replacement, the paging system sees a page that is in memory but has not been referenced or modified: it is the page that the low-priority process just brought in. This page looks like a perfect replacement. It is clean and will not need to be written out, and it apparently has not been used for a long time.

Whether the high-priority process should be able to replace the low-priority process is a policy decision. After all, we are simply delaying the low-priority process for the benefit of the high-priority process. However, we are wasting the effort spent to bring in the page for the low-priority process. If we decide to prevent replacement of a newly brought-in page until it can be used at least once, then we can use the lock bit to implement this mechanism. When a page is selected for replacement, its lock bit is turned on. It remains on until the faulting process is again dispatched.

Using a lock bit can be dangerous: the lock bit may get turned on but never turned off. Should this situation occur (because of a bug in the operating system, for example), the locked frame becomes unusable. For instance, Solaris allows locking "hints," but it is free to disregard these hints if the free-frame pool becomes too small or if an individual process requests that too many pages be locked in memory.

10.10 Operating-System Examples

In this section, we describe how Linux, Windows and Solaris manage virtual memory.

10.10.1 Linux

In Section 10.8.2, we discussed how Linux manages kernel memory using slab allocation. We now cover how Linux manages virtual memory. Linux uses demand paging, allocating pages from a list of free frames. In addition, it uses a global page-replacement policy similar to the LRU-approximation clock algorithm described in Section 10.4.5.2. To manage memory, Linux maintains two types of page lists: an `active_list` and an `inactive_list`. The `active_list` contains the pages that are considered in use, while the `inactive_list` contains pages that have not recently been referenced and are eligible to be reclaimed.

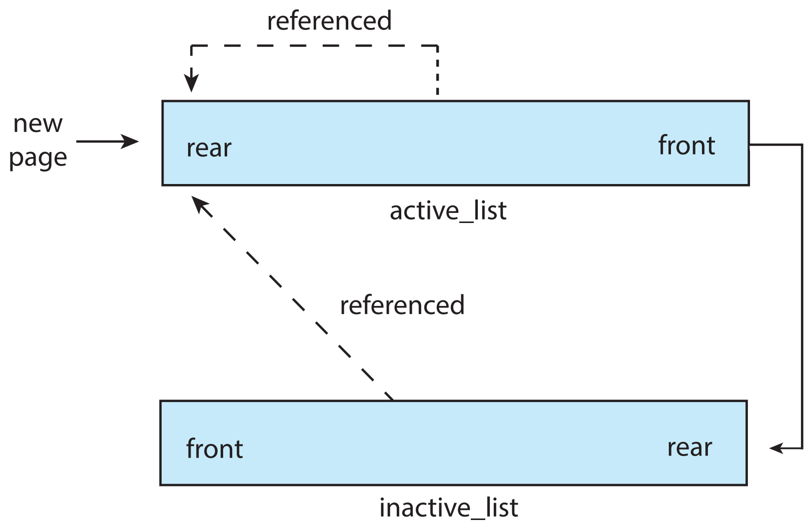


Figure 10.29 The Linux `active_list` and `inactive_list` structures.

Each page has an ***accessed*** bit that is set whenever the page is referenced. (The actual bits used to mark page access vary by architecture.) When a page is first allocated, its accessed bit is set, and it is added to the rear of the `active_list`. Similarly, whenever a page in the `active_list` is referenced, its accessed bit is set, and the page moves to the rear of the list. Periodically, the accessed bits for pages in the `active_list` are reset. Over time, the least recently used page will be at the front of the `active_list`. From there, it may migrate to the rear of the `inactive_list`. If a page in the `inactive_list` is referenced, it moves back to the rear of the `active_list`. This pattern is illustrated in Figure 10.29.

The two lists are kept in relative balance, and when the `active_list` grows much larger than the `inactive_list`, pages at the front of the `active_list` move to the `inactive_list`, where they become eligible for reclamation. The Linux kernel has a page-out daemon process `kswapd` that periodically awakens and checks the amount of free memory in the system. If free memory falls below a certain threshold, `kswapd` begins scanning pages in the `inactive_list` and reclaiming them for the free list. Linux virtual memory management is discussed in greater detail in Chapter 20.

10.10.2 Windows

Windows 10 supports 32- and 64-bit systems running on Intel (IA-32 and x86-64) and ARM architectures. On 32-bit systems, the default virtual address space of a process is 2 GB, although it can be extended to 3 GB. 32-bit systems support 4 GB of physical memory. On 64-bit systems, Windows 10 has a 128-TB virtual address space and supports up to 24 TB of physical memory. (Versions of Windows Server support up to 128 TB of physical memory.) Windows 10 implements most of the memory-management features described thus far, including shared libraries, demand paging, copy-on-write, paging, and memory compression.

Windows 10 implements virtual memory using demand paging with **clustering**, a strategy that recognizes locality of memory references and therefore handles page faults by bringing in not only the faulting page but also several pages immediately preceding and following the faulting page. The size of a cluster varies by page type. For a data page, a cluster contains three pages(the page before and the page after the faulting page); all other page faults have a cluster size of seven.

A key component of virtual memory management in Windows 10 is working-set management. When a process is created, it is assigned a working-set minimum of 50 pages and a working-set maximum of 345 pages. The **working-set minimum** is the minimum number of pages the process is guaranteed to have in memory; if sufficient memory is available, a process may be assigned as many pages as its **working-set maximum**. Unless a process is configured with **hard working-set limits**, these values may be ignored. A process can grow beyond its working-set maximum if sufficient memory is available. Similarly, the amount of memory allocated to a process can shrink below the minimum in periods of high demand for memory.

Windows uses the LRU-approximation clock algorithm, as described in Section 10.4.5.2, with a combination of local and global page-replacement policies. The virtual memory manager maintains a list of free page frames. Associated with this list is a threshold value that indicates whether sufficient free memory is available. If a page fault occurs for a process that is below its working-set maximum, the virtual memory manager allocates a page from the list of free pages. If a process that is at its working-set maximum incurs a page fault and sufficient memory is available, the process is allocated a free page, which allows it to grow beyond its working-set maximum. If the amount of free memory is insufficient, however, the kernel must select a page from the process's working set for replacement using a local LRU page-replacement policy.

When the amount of free memory falls below the threshold, the virtual memory manager uses a global replacement tactic known as **automatic working-set trimming** to restore the value to a level above the threshold. Automatic working-set trimming works by evaluating the number of pages allocated to processes. If a process has been allocated more pages than its working-set minimum, the virtual memory manager removes pages from the working set until either there is sufficient memory available or the process has reached its working-set minimum. Larger processes that have been idle are targeted before smaller, active processes. The trimming procedure continues until there is sufficient free memory, even if it is necessary to remove pages from a process already below its working set minimum. Windows performs working-set trimming on both user-mode and system processes.

10.10.3 Solaris

In Solaris, when a thread incurs a page fault, the kernel assigns a page to the faulting thread from the list of free pages it maintains. Therefore, it is imperative that the kernel keep a sufficient amount of free memory available. Associated with this list of free pages is a parameter—`lotsfree`—that represents a threshold to begin paging. The `lotsfree` parameter is typically set to 1/64 the size of the physical memory. Four times per second, the kernel checks whether the amount of free memory is less than `lotsfree`. If the number of free pages falls below lotsfree, a process known as a **pageout** starts up. The pageout process is similar to the second-chance algorithm described in Section 10.4.5.2, except that it uses two hands while scanning pages, rather than one.

The pageout process works as follows: The front hand of the clock scans all pages in memory, setting the reference bit to 0. Later, the back hand of the clock examines the reference bit for the pages in memory, appending each page whose reference bit is still set to 0 to the free list and writing its contents to secondary storage if it has been modified. Solaris also manages minor page faults by allowing a process to reclaim a page from the free list if the page is accessed before being reassigned to another process.

The pageout algorithm uses several parameters to control the rate at which pages are scanned (known as the scanrate). The scanrate is expressed in pages per second and ranges from slowscan to fastscan. When free memory falls below lotsfree, scanning occurs at slowscan pages per second and progresses to fastscan, depending on the amount of free memory available. The default value of slowscan is 100 pages per second. Fastscan is typically set to the value (total physical pages)/2 pages per second, with a maximum of 8,192 pages per second. This is shown in Figure 10.30 (with fastscan set to the maximum).

The distance (in pages) between the hands of the clock is determined by a system parameter, handspread. The amount of time between the front hand's clearing a bit and the back hand's investigating its value depends on the scanrate and the handspread. If scanrate is 100 pages per second and handspread is 1,024 pages, 10 seconds can pass between the time a bit is set by the front hand and the time it is checked by the back hand. However, because of the demands placed on the memory system, a scanrate of several thousand is not uncommon. This means that the amount of time between clearing and investigating a bit is often a few seconds.

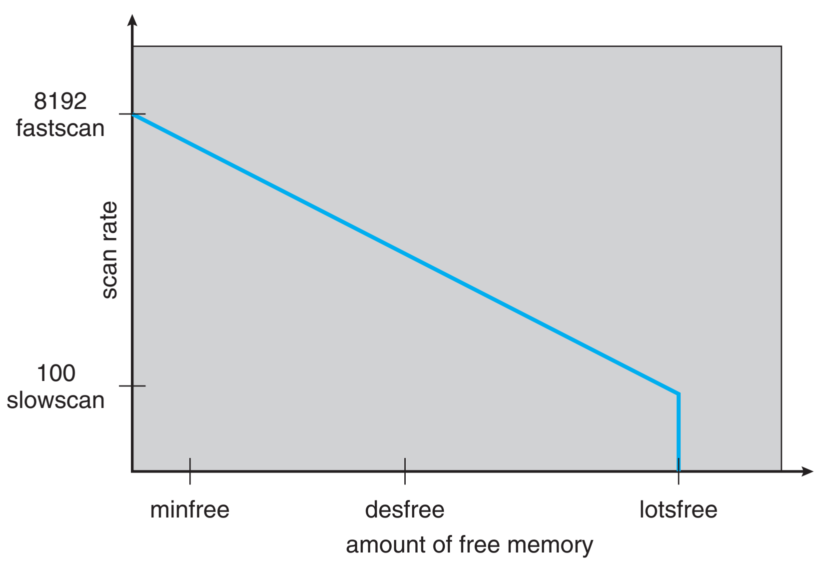


Figure 10.30 Solaris page scanner.

As mentioned above, the pageout process checks memory four times per second. However, if free memory falls below the value of `desfree` (the desired amount of free memory in the system), pageout will run a hundred times per second with the intention of keeping at least `desfree` free memory available (Figure 10.30). If the pageout process is unable to keep the amount of free memory at `desfree` for a 30-second average, the kernel begins swapping processes, thereby freeing all pages allocated to swapped processes. In general, the kernel looks for processes that have been idle for long periods of time. If the system is unable to maintain the amount of free memory at `minfree`, the pageout process is called for every request for a new page.

The page-scanning algorithm skips pages belonging to libraries that are being shared by several processes, even if they are eligible to be claimed by the scanner. The algorithm also distinguishes between pages allocated to processes and pages allocated to regular data files. This is known as **priority paging** and is covered in Section 14.6.2.

10.11 Summary

- Virtual memory abstracts physical memory into an extremely large uniform array of storage.
- The benefits of virtual memory include the following: (1) a program can be larger than physical memory, (2) a program does not need to be entirely in memory, (3) processes can share memory, and (4) processes can be created more efficiently.
- Demand paging is a technique whereby pages are loaded only when they are demanded during program execution. Pages that are never demanded are thus never loaded into memory.
- A page fault occurs when a page that is currently not in memory is accessed. The page must be brought from the backing store into an available page frame in memory.
- Copy-on-write allows a child process to share the same address space as its parent. If either the child or the parent process writes (modifies) a page, a copy of the page is made.
- When available memory runs low, a page-replacement algorithm selects an existing page in memory to replace with a new page. Page-replacement algorithms include FIFO, optimal, and LRU. Pure LRU algorithms are impractical to implement, and most systems instead use LRU-approximation algorithms.
- Global page-replacement algorithms select a page from any process in the system for replacement, while local page-replacement algorithms select a page from the faulting process.
- Thrashing occurs when a system spends more time paging than executing.
- A locality represents a set of pages that are actively used together. As a process executes, it moves from locality to locality. A working set is based on locality and is defined as the set of pages currently in use by a process.

- Memory compression is a memory-management technique that compresses a number of pages into a single page. Compressed memory is an alternative to paging and is used on mobile systems that do not support paging.
- Kernel memory is allocated differently than user-mode processes; it is allocated in contiguous chunks of varying sizes. Two common techniques for allocating kernel memory are (1) the buddy system and (2) slab allocation.
- TLB reach refers to the amount of memory accessible from the TLB and is equal to the number of entries in the TLB multiplied by the page size. One technique for increasing TLB reach is to increase the size of pages.
- Linux, Windows, and Solaris manage virtual memory similarly, using demand paging and copy-on-write, among other features. Each system also uses a variation of LRU approximation known as the clock algorithm.

Practice Exercises

10.1 Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

10.2 Assume that you have a page-reference string for a process with m frames (initially all empty). The page-reference string has length p, and n distinct page numbers occur in it. Answer these questions for any page-replacement algorithms:

a. What is a lower bound on the number of page faults?

b. What is an upper bound on the number of page faults?

10.3 Consider the following page-replacement algorithms. Rank these algorithms on a five-point scale from "bad" to "perfect" according to their page-fault rate. Separate those algorithms that suffer from Belady's anomaly from those that do not.

a. LRU replacement

b. FIFO replacement

c. Optimal replacement

d. Second-chance replacement

10.4 An operating system supports a paged virtual memory. The central processor has a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1,000 words, and the paging device is a drum that rotates at 3,000 revolutions per minute and transfers 1 million words per second. The following statistical measurements were obtained from the system:

- One percent of all instructions executed accessed a page other than the current page.
- Of the instructions that accessed another page, 80 percent accessed a page already in memory.

- When a new page was required, the replaced page was modified 50 percent of the time.

Calculate the effective instruction time on this system, assuming that the system is running one process only and that the processor is idle during drum transfers.

10.5 Consider the page table for a system with 12-bit virtual and physical addresses and 256-byte pages.

Page	Page Frame
0	–
1	2
2	C
3	A
4	–
5	4
6	3
7	–
8	B
9	0

The list of free page frames is *D*, *E*, *F* (that is, *D* is at the head of the list, *E* is second, and *F* is last). A dash for a page frame indicates that the page is not in memory.

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal.

- 9EF
- 111
- 700
- 0FF

10.6 Discuss the hardware functions required to support demand paging.

10.7 Consider the two-dimensional array `A`:

```
int A[][] = new int[100][100];
```

where `A[0][0]` is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (locations 0 to 199). Thus, every instruction fetch will be from page 0.

For three page frames, how many page faults are generated by the following array-initialization loops? Use LRU replacement, and assume that page frame 1 contains the process and the other two are initially empty.

a.
```
for (int j = 0; j < 100; j++)
    for (int i = 0; i < 100; i++)
        A[i][j] = 0;
```

b.
```
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
        A[i][j] = 0;
```

10.8 Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement

10.9 Consider the following page reference string:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0 , 1.

Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

- LRU replacement
- FIFO replacement
- Optimal replacement

10.10 Suppose that you want to use a paging algorithm that requires a reference bit (such as second-chance replacement or working-set model), but the hardware does not provide one. Sketch how you could simulate a reference bit even if one were not provided by the hardware, or explain why it is not possible to do so. If it is possible, calculate what the cost would be.

10.11 You have devised a new page-replacement algorithm that you think may be optimal. In some contorted test cases, Belady's anomaly occurs. Is the new algorithm optimal? Explain your answer.

10.12 Segmentation is similar to paging but uses variable-sized "pages." Define two segment-replacement algorithms, one based on the FIFO page-replacement scheme and the other on the LRU page-replacement scheme. Remember that since segments are not the same size, the segment that is chosen for replacement may be too small to leave enough consecutive locations for the needed segment. Consider strategies for systems where segments cannot be relocated and strategies for systems where they can.

10.13 Consider a demand-paged computer system where the degree of multiprogramming is currently fixed at four. The system was recently measured to determine utilization of the CPU and the paging disk. Three alternative results are shown below. For each case, what is happening? Can the degree of multiprogramming be increased to increase the CPU utilization? Is the paging helping?

a. CPU utilization 13 percent; disk utilization 97 percent

b. CPU utilization 87 percent; disk utilization 3 percent

c. CPU utilization 13 percent; disk utilization 3 percent

10.14 We have an operating system for a machine that uses base and limit registers, but we have modified the machine to provide a page table. Can the page table be set up to simulate base and limit registers? How can it be, or why can it not be?

Further Reading

The working-set model was developed by [Denning (1968)]. The enhanced clock algorithm is discussed by [Carr and Hennessy (1981)]. [Russinovich et al. (2017)] describe how Windows implements virtual memory and memory compression. Compressed memory in Windows 10 is further discussed in http://www.makeuseof.com/tag/ram-compression-improves-memory-responsiveness-windows-10.

[McDougall and Mauro (2007)] discuss virtual memory in Solaris. Virtual memory techniques in Linux are described in [Love (2010)] and [Mauerer (2008)]. FreeBSD is described in [McKusick et al. (2015)].

Bibliography

[Carr and Hennessy (1981)] W. R. Carr and J. L. Hennessy, "WSClock—A Simple and Effective Algorithm for Virtual Memory Management", *Proceedings of the ACM Symposium on Operating Systems Principles* (1981), pages 87–95.

[Denning (1968)] P. J. Denning, "The Working Set Model for Program Behavior", *Communications of the ACM*, Volume 11, Number 5 (1968), pages 323–333.

[Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

[Mauerer (2008)] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

[McDougall and Mauro (2007)] R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).

[McKusick et al. (2015)] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System–Second Edition*, Pearson (2015).

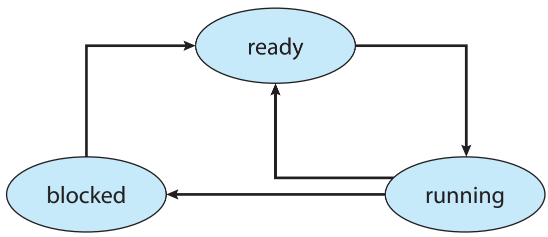
[Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1,* Seventh Edition, Microsoft Press (2017).

Chapter 10 Exercises

10.15 Assume that a program has just referenced an address in virtual memory. Describe a scenario in which each of the following can occur. (If no such scenario can occur, explain why.)

- TLB miss with no page fault
- TLB miss with page fault
- TLB hit with no page fault
- TLB hit with page fault

10.16 A simplified view of thread states is ***ready***, ***running***, and ***blocked***, where a thread is either ready and waiting to be scheduled, is running on the processor, or is blocked (for example, waiting for I/O).



Assuming a thread is in the ***running*** state, answer the following questions, and explain your answers:

a. Will the thread change state if it incurs a page fault? If so, to what state will it change?

b. Will the thread change state if it generates a TLB miss that is resolved in the page table? If so, to what state will it change?

c. Will the thread change state if an address reference is resolved in the page table? If so, to what state will it change?

10.17 Consider a system that uses pure demand paging.

a. When a process first starts execution, how would you characterize the page-fault rate?

b. Once the working set for a process is loaded into memory, how would you characterize the page-fault rate?

c. Assume that a process changes its locality and the size of the new working set is too large to be stored in available free memory. Identify some options system designers could choose from to handle this situation.

10.18 The following is a page table for a system with 12-bit virtual and physical addresses and 256-byte pages. Free page frames are to be allocated in the order `9, F, D`. A dash for a page frame indicates that the page is not in memory.

Page	Page Frame
0	0 x 4
1	0 x B
2	0 x A
3	–
4	–
5	0 x 2
6	–
7	0 x 0
8	0 x C
9	0 x 1

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal. In the case of a page fault, you must use one of the free frames to update the page table and resolve the logical address to its corresponding physical address.

- `0x2A1`
- `0x4E6`
- `0x94A`
- `0x316`

10.19 What is the copy-on-write feature, and under what circumstances is its use beneficial? What hardware support is required to implement this feature?

10.20 A certain computer provides its users with a virtual memory space of 2^{32} bytes. The computer has 2^{22} bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4,096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.

10.21 Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds.

Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?

10.22 Consider the page table for a system with 16-bit virtual and physical addresses and 4,096-byte pages.

Page	Page Frame	Reference Bit
0	9	0
1	–	0
2	10	0
3	15	0
4	6	0
5	13	0
6	8	0
7	12	0
8	7	0
9	–	0
10	5	0
11	4	0
12	1	0
13	0	0
14	–	0
15	2	0

The reference bit for a page is set to 1 when the page has been referenced. Periodically, a thread zeroes out all values of the reference bit. A dash for a page frame indicates that the page is not in memory. The page-replacement algorithm is localized LRU, and all numbers are provided in decimal.

a. Convert the following virtual addresses (in hexadecimal) to the equivalent physical addresses. You may provide answers in either hexadecimal or decimal. Also set the reference bit for the appropriate entry in the page table.

- `0x621C`
- `0xF0A3`
- `0xBC1A`
- `0x5BAA`
- `0x0BA1`

b. Using the above addresses as a guide, provide an example of a logical address (in hexadecimal) that results in a page fault.

c. From what set of page frames will the LRU page-replacement algorithm choose in resolving a page fault?

10.23 When a page fault occurs, the process requesting the page must block while waiting for the page to be brought from disk into physical memory. Assume that there exists a process with five user-level threads and that the mapping of user threads to kernel threads is many to one. If one user thread incurs a page fault while accessing its stack, would the other user threads belonging to the same process also be affected by the page fault—that is, would they also have to wait for the faulting page to be brought into memory? Explain.

10.24 Apply the (1) FIFO, (2) LRU, and (3) optimal (OPT) replacement algorithms for the following page-reference strings:

- 2, 6, 9, 2, 4, 2, 1, 7, 3, 0, 5, 2, 1, 2, 9, 5, 7, 3, 8, 5
- 0, 6, 3, 0, 2, 6, 3, 5, 2, 4, 1, 3, 0, 6, 1, 4, 2, 3, 5, 7
- 3, 1, 4, 2, 5, 4, 1, 3, 5, 2, 0, 1, 1, 0, 2, 3, 4, 5, 0, 1
- 4, 2, 1, 7, 9, 8, 3, 5, 2, 6, 8, 1, 0, 7, 2, 4, 1, 3, 5, 8
- 0, 1, 2, 3, 4, 4, 3, 2, 1, 0, 0, 1, 2, 3, 4, 4, 3, 2, 1, 0

Indicate the number of page faults for each algorithm assuming demand paging with three frames.

10.25 Assume that you are monitoring the rate at which the pointer in the clock algorithm moves. (The pointer indicates the candidate page for replacement.) What can you say about the system if you notice the following behavior:

a. Pointer is moving fast.

b. Pointer is moving slow.

10.26 Discuss situations in which the least frequently used (LFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

10.27 Discuss situations in which the most frequently used (MFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

10.28 The KHIE (pronounced “k-hi”) operating system uses a FIFO replacement algorithm for resident pages and a free-frame pool of recently used pages. Assume that the free-frame pool is managed using the LRU replacement policy. Answer the following questions:

a. If a page fault occurs and the page does not exist in the free-frame pool, how is free space generated for the newly requested page?

b. If a page fault occurs and the page exists in the free-frame pool, how are the resident page set and the free-frame pool managed to make space for the requested page?

c. To what does the system degenerate if the number of resident pages is set to one?

d. To what does the system degenerate if the number of pages in the free-frame pool is zero?

10.29 Consider a demand-paging system with the following time-measured utilizations:

CPU utilization	20%
Paging disk	97.7%
Other I/O devices	5%

For each of the following, indicate whether it will (or is likely to) improve CPU utilization. Explain your answers.

a. Install a faster CPU.

b. Install a bigger paging disk.

c. Increase the degree of multiprogramming.

d. Decrease the degree of multiprogramming.

e. Install more main memory.

f. Install a faster hard disk or multiple controllers with multiple hard disks.

g. Add prepaging to the page-fetch algorithms.

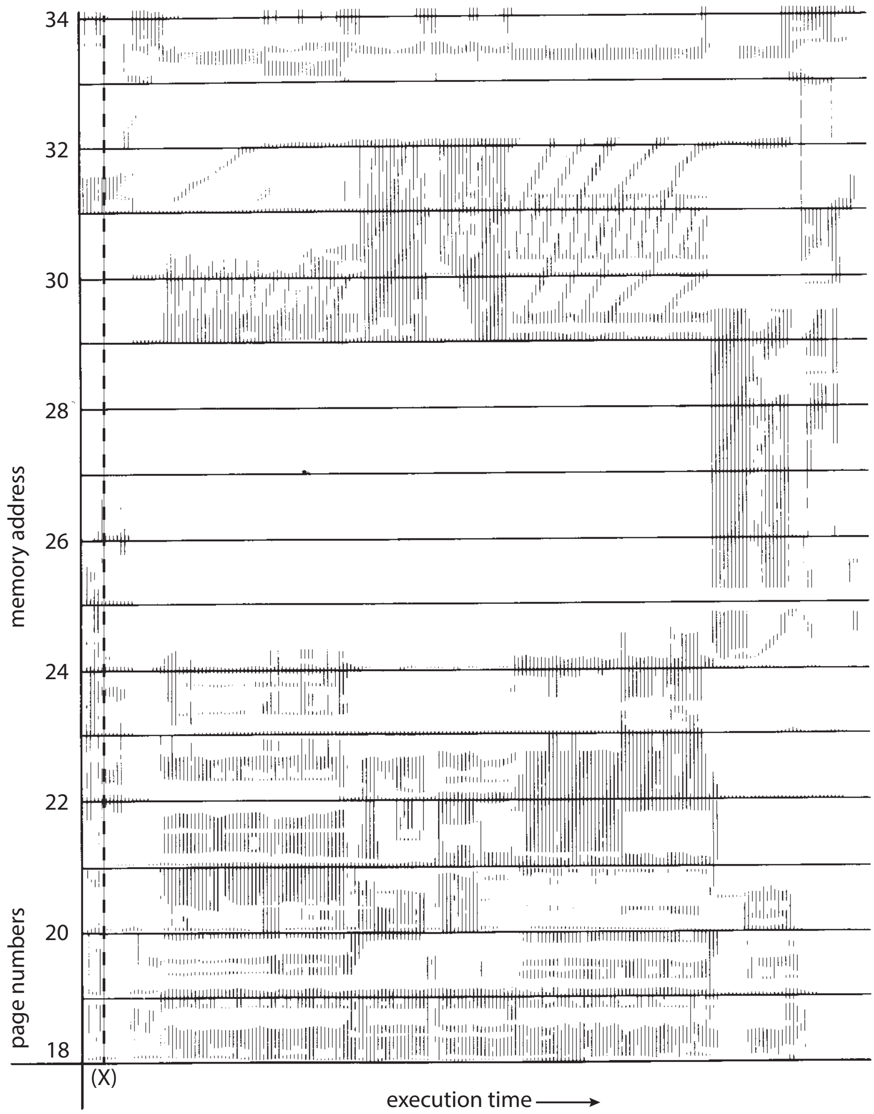
h. Increase the page size.

10.30 Explain why minor page faults take less time to resolve than major page faults.

10.31 Explain why compressed memory is used in operating systems for mobile devices.

10.32 Suppose that a machine provides instructions that can access memory locations using the one-level indirect addressing scheme. What sequence of page faults is incurred when all of the pages of a program are currently nonresident and the first instruction of the program is an indirect memory-load operation? What happens when the operating system is using a per-process frame allocation technique and only two pages are allocated to this process?

10.33 Consider the page references:



What pages represent the locality at time (X)?

10.34 Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discard that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?

10.35 A page-replacement algorithm should minimize the number of page faults. We can achieve this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages associated with that frame. Then, to replace a page, we can search for the page frame with the smallest counter.

a. Define a page-replacement algorithm using this basic idea. Specifically address these problems:

   - What is the initial value of the counters?
   - When are counters increased?
   - When are counters decreased?
   - How is the page to be replaced selected?

b. How many page faults occur for your algorithm for the following reference string with four page frames?

   1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

c. What is the minimum number of page faults for an optimal page-replacement strategy for the reference string in part b with four page frames?

10.36 Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference if the page-table entry is in the associative memory.

Assume that 80 percent of the accesses are in the associative memory and that, of those remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?

10.37 What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

10.38 Is it possible for a process to have two working sets, one representing data and another representing code? Explain.

10.39 Consider the parameter Δ used to define the working-set window in the working-set model. When Δ is set to a low value, what is the effect on the page-fault frequency and the number of active (nonsuspended) processes currently executing in the system? What is the effect when Δ is set to a very high value?

10.40 In a 1,024-KB segment, memory is allocated using the buddy system. Using Figure 10.26 as a guide, draw a tree illustrating how the following memory requests are allocated:

- Request 5-KB
- Request 135 KB.
- Request 14 KB.
- Request 3 KB.

- Request 12 KB.

Next, modify the tree for the following releases of memory. Perform coalescing whenever possible:

- Release 3 KB.
- Release 5 KB.
- Release 14 KB.
- Release 12 KB.

10.41 A system provides support for user-level and kernel-level threads. The mapping in this system is one to one (there is a corresponding kernel thread for each user thread). Does a multithreaded process consist of (a) a working set for the entire process or (b) a working set for each thread? Explain

10.42 The slab-allocation algorithm uses a separate cache for each different object type. Assuming there is one cache per object type, explain why this scheme doesn't scale well with multiple CPUs. What could be done to address this scalability issue?

10.43 Consider a system that allocates pages of different sizes to its processes. What are the advantages of such a paging scheme? What modifications to the virtual memory system would be needed to provide this functionality?

Programming Problems

10.44 Write a program that implements the FIFO, LRU, and optimal (OPT) page-replacement algorithms presented in Section 10.4. Have your program initially generate a random page-reference string where page numbers range from 0 to 9. Apply the random page-reference string to each algorithm, and record the number of page faults incurred by each algorithm. Pass the number of page frames to the program at startup. You may implement this program in any programming language of your choice. (You may find your implementation of either FIFO or LRU to be helpful in the virtual memory manager programming project.)

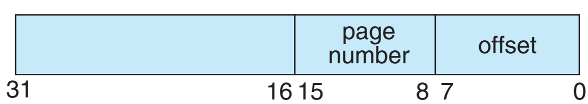
Programming Projects

Designing a Virtual Memory Manager

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size $2^{16} = 65{,}536$ bytes. Your program will read from a file containing logical addresses and, using a TLB and a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. Your learning goal is to use simulation to understand the steps involved in translating logical to physical addresses. This will include resolving page faults using demand paging, managing a TLB, and implementing a page-replacement algorithm.

Specific

Your program will read a file containing several 32-bit integer numbers that represent logical addresses. However, you need only be concerned with 16-bit addresses, so you must mask the rightmost 16 bits of each logical address. These 16 bits are divided into (1) an 8-bit page number and (2) an 8-bit page offset. Hence, the addresses are structured as shown as:



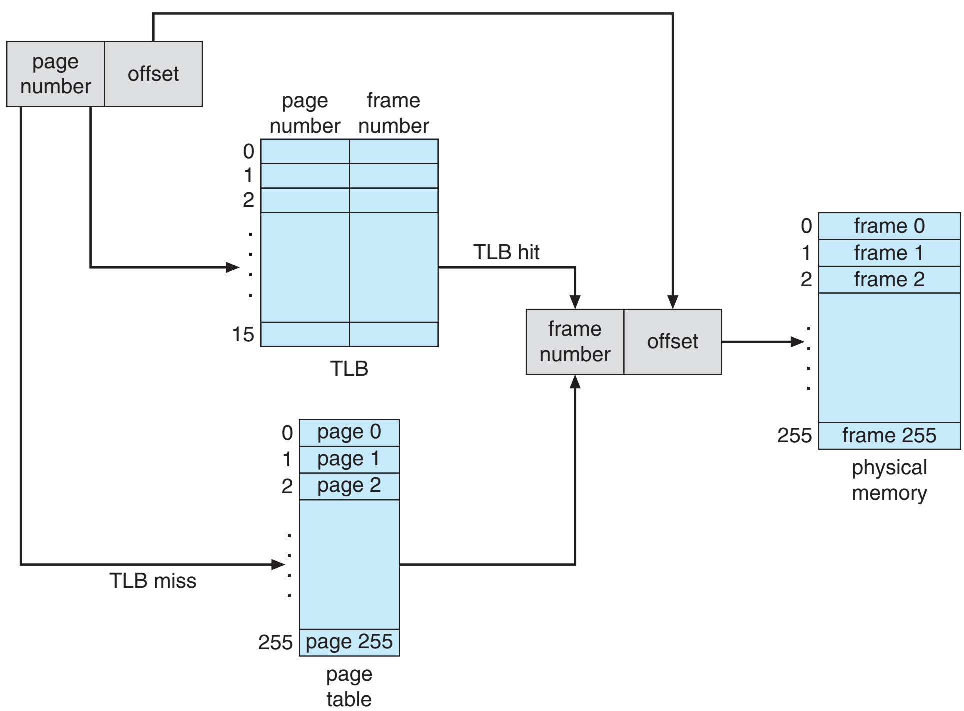
Other specifics include the following:

- 2^8 entries in the page table
- Page size of 2^8 bytes
- 16 entries in the TLB
- Frame size of 2^8 bytes
- 256 frames
- Physical memory of 65,536 bytes (256 frames × 256-byte frame size)

Additionally, your program need only be concerned with reading logical addresses and translating them to their corresponding physical addresses. You do not need to support writing to the logical address space.

Address Translation

Your program will translate logical to physical addresses using a TLB and page table as outlined in Section 9.3. First, the page number is extracted from the logical address, and the TLB is consulted. In the case of a TLB hit, the frame number is obtained from the TLB. In the case of a TLB miss, the page table must be consulted. In the latter case, either the frame number is obtained from the page table, or a page fault occurs. A visual representation of the address-translation process is:



Handling Page Faults

Your program will implement demand paging as described in Section 10.2. The backing store is represented by the file `BACKING_STORE.bin`, a binary file of size 65,536 bytes. When a page fault occurs, you will read in a 256-byte page from the file `BACKING_STORE` and store it in an available page frame in physical memory. For example, if a logical address with page number 15 resulted in a page fault, your program would read in page 15 from `BACKING_STORE` (remember that pages begin at 0 and are 256 bytes in size) and store it in a page frame in physical memory. Once this frame is stored (and the page table and TLB are updated), subsequent accesses to page 15 will be resolved by either the TLB or the page table.

You will need to treat BACKING_STORE.bin as a random-access file so that you can randomly seek to certain positions of the file for reading. We suggest using the standard C library functions for performing I/O, including `fopen()`, `fread()`, `fseek()`, and `fclose()`.

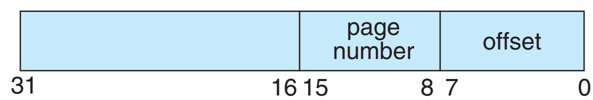
The size of physical memory is the same as the size of the virtual address space—65,536 bytes—so you do not need to be concerned about page replacements during a page fault. Later, we describe a modification to this project using a smaller amount of physical memory; at that point, a page-replacement strategy will be required.

Test File

We provide the file `addresses.txt`, which contains integer values representing logical addresses ranging from $0 to 65535$ (the size of the virtual address space). Your program will open this file, read each logical address and translate it to its corresponding physical address, and output the value of the signed byte at the physical address.

How to Begin

First, write a simple program that extracts the page number and offset based on:



from the following integer numbers:

$$1, 256, 32768, 32769, 128, 65534, 33153$$

Perhaps the easiest way to do this is by using the operators for bit-masking and bit-shifting. Once you can correctly establish the page number and offset from an integer number, you are ready to begin.

Initially, we suggest that you bypass the TLB and use only a page table. You can integrate the TLB once your page table is working properly. Remember, address translation can work without a TLB; the TLB just makes it faster. When you are ready to implement the TLB, recall that it has only sixteen entries, so you will need to use a replacement strategy when you update a full TLB. You may use either a FIFO or an LRU policy for updating your TLB.

How to Run Your Program

Your program should run as follows:

```
./a.out addresses.txt
```

Your program will read in the file `addresses.txt`, which contains 1,000 logical addresses ranging from 0 to 65535. Your program is to translate each logical address to a physical address and determine the contents of the signed byte stored at the correct physical address. (Recall that in the C language, the `char` data type occupies a byte of storage, so we suggest using `char` values.)

Your program is to output the following values:

1. The logical address being translated (the integer value being read from `addresses.txt`).
2. The corresponding physical address (what your program translates the logical address to).
3. The signed byte value stored in physical memory at the translated physical address.

We also provide the file `correct.txt`, which contains the correct output values for the file `addresses.txt`. You should use this file to determine if your program is correctly translating logical to physical addresses.

Statistics

After completion, your program is to report the following statistics:

1. Page-fault rate—The percentage of address references that resulted in page faults.
2. TLB hit rate—The percentage of address references that were resolved in the TLB.

Since the logical addresses in `addresses.txt` were generated randomly and do not reflect any memory access locality, do not expect to have a high TLB hit rate.

Page Replacement

Thus far, this project has assumed that physical memory is the same size as the virtual address space. In practice, physical memory is typically much smaller than a virtual address space. This phase of the project now assumes using a smaller physical address space with 128 page frames rather than 256. This change will require modifying your program so that it keeps track of free page frames as well as implementing a page-replacement policy using either FIFO or LRU (Section 10.4) to resolve page faults when there is no free memory.

Part Five

Storage Management

Computer systems must provide mass storage for permanently storing files and data. Modern computers implement mass storage as secondary storage, using both hard disks and nonvolatile memory devices.

Secondary storage devices vary in many aspects. Some transfer a character at a time, and some a block of characters. Some can be accessed only sequentially, and others randomly. Some transfer data synchronously, and others asynchronously. Some are dedicated, and some shared. They can be read-only or read–write. And although they vary greatly in speed, they are in many ways the slowest major component of the computer.

Because of all this device variation, the operating system needs to provide a wide range of functionality so that applications can control all aspects of the devices. One key goal of an operating system's I/O subsystem is to provide the simplest interface possible to the rest of the system. Because devices are a performance bottleneck, another key is to optimize I/O for maximum concurrency.

CHAPTER 11

Mass-Storage Structure

In this chapter, we discuss how mass storage—the nonvolatile storage system of a computer—is structured. The main mass-storage system in modern computers is secondary storage, which is usually provided by hard disk drives (HDD) and nonvolatile memory (NVM) devices. Some systems also have slower, larger, tertiary storage, generally consisting of magnetic tape, optical disks, or even cloud storage.

Because the most common and important storage devices in modern computer systems are HDDs and NVM devices, the bulk of this chapter is devoted to discussing these two types of storage. We first describe their physical structure. We then consider scheduling algorithms, which schedule the order of I/Os to maximize performance. Next, we discuss device formatting and management of boot blocks, damaged blocks, and swap space. Finally, we examine the structure of RAID systems.

There are many types of mass storage, and we use the general term ***nonvolatile storage*** (NVS) or talk about storage "drives" when the discussion includes all types. Particular devices, such as HDDs and NVM devices, are specified as appropriate.

CHAPTER OBJECTIVES

- Describe the physical structures of various secondary storage devices and the effect of a device's structure on its uses.
- Explain the performance characteristics of mass-storage devices.
- Evaluate I/O scheduling algorithms.
- Discuss operating-system services provided for mass storage, including RAID.

11.1 Overview of Mass-Storage Structure

The bulk of secondary storage for modern computers is provided by **hard disk drives** (**HDDs**) and **nonvolatile memory** (**NVM**) devices. In this section,

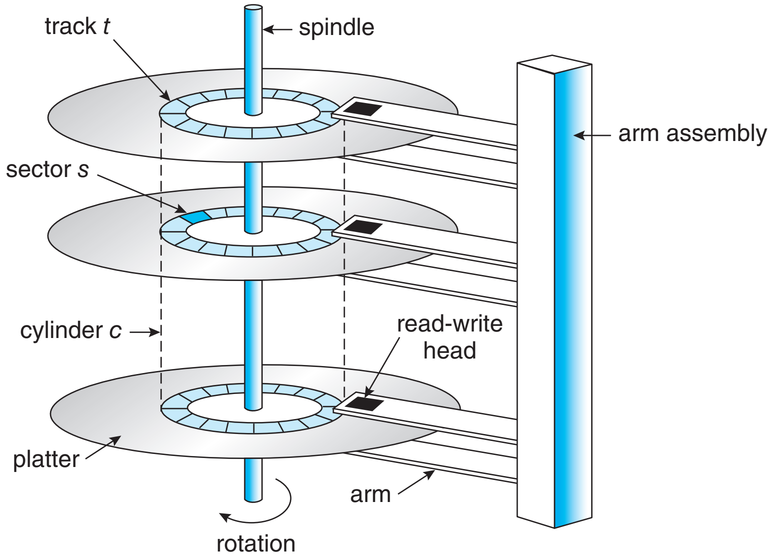


Figure 11.1 HDD moving-head disk mechanism.

we describe the basic mechanisms of these devices and explain how operating systems translate their physical properties to logical storage via address mapping.

11.1.1 Hard Disk Drives

Conceptually, HDDs are relatively simple (Figure 11.1). Each disk **platter** has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters, and we read information by detecting the magnetic pattern on the platters.

A read–write head "flies" just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks**, which are subdivided into **sectors**. The set of tracks at a given arm position make up a **cylinder**. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. Each sector has a fixed size and is the smallest unit of transfer. The sector size was commonly 512 bytes until around 2010. At that point, many manufacturers start migrating to 4KB sectors. The storage capacity of common disk drives is measured in gigabytes and terabytes. A disk drive with the cover removed is shown in Figure 11.2.

A disk drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute (**RPM**). Common drives spin at 5,400, 7,200, 10,000, and 15,000 RPM. Some drives power down when not in use and spin up upon receiving an I/O request. Rotation speed relates to transfer rates. The **transfer rate** is the rate at which data flow between the drive and the computer. Another performance aspect, the **positioning time**, or **random-access time**, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the

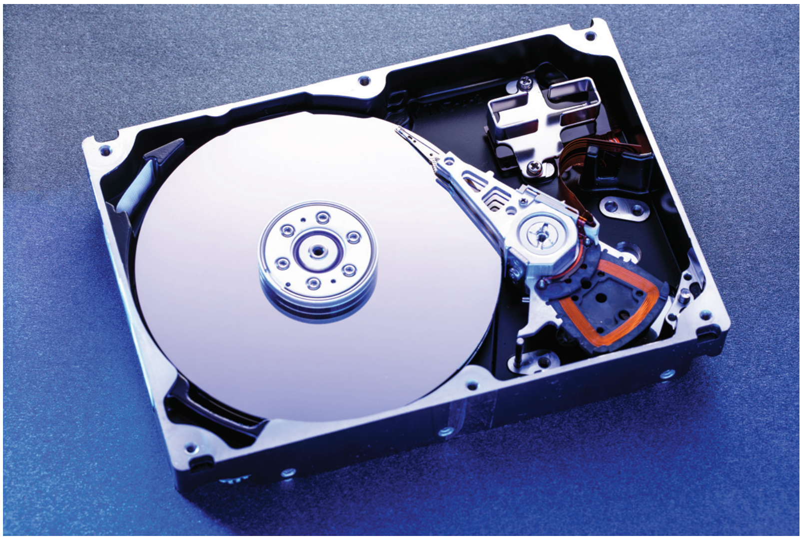


Figure 11.2 A 3.5-inch HDD with cover removed.

desired sector to rotate to the disk head, called the **rotational latency**. Typical disks can transfer tens to hundreds of megabytes of data per second, and they have seek times and rotational latencies of several milliseconds. They increase performance by having DRAM buffers in the drive controller.

The disk head flies on an extremely thin cushion (measured in microns) of air or another gas, such as helium, and there is a danger that the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, the head will sometimes damage the magnetic surface. This accident is called a **head crash**. A head crash normally cannot be repaired; the entire disk must be replaced, and the data on the disk are lost unless they were backed up to other storage or RAID protected. (RAID is discussed in Section 11.8.)

HDDs are sealed units, and some chassis that hold HDDs allow their removal without shutting down the system or storage chassis. This is helpful when a system needs more storage than can be connected at a given time or when it is necessary to replace a bad drive with a working one. Other types of storage media are also **removable**, including CDs, DVDs, and Blu-ray discs.

DISK TRANSFER RATES

As with many aspects of computing, published performance numbers for disks are not the same as real-world performance numbers. Stated transfer rates are always higher than **effective transfer rates**, for example. The transfer rate may be the rate at which bits can be read from the magnetic media by the disk head, but that is different from the rate at which blocks are delivered to the operating system.

11.1.2 Nonvolatile Memory Devices

Nonvolatile memory (NVM) devices are growing in importance. Simply described, NVM devices are electrical rather than mechanical. Most commonly, such a device is composed of a controller and flash NAND die semiconductor chips, which are used to store data. Other NVM technologies exist, like DRAM with battery backing so it doesn't lose its contents, as well as other semiconductor technology like 3D XPoint, but they are far less common and so are not discussed in this book.

11.1.2.1 Overview of Nonvolatile Memory Devices

Flash-memory-based NVM is frequently used in a disk-drive-like container, in which case it is called a **solid-state disk** (**SSD**) (Figure 11.3). In other instances, it takes the form of a **USB drive** (also known as a thumb drive or flash drive) or a DRAM stick. It is also surface-mounted onto motherboards as the main storage in devices like smartphones. In all forms, it acts and can be treated in the same way. Our discussion of NVM devices focuses on this technology.

NVM devices can be more reliable than HDDs because they have no moving parts and can be faster because they have no seek time or rotational latency. In addition, they consume less power. On the negative side, they are more expensive per megabyte than traditional hard disks and have less capacity than the larger hard disks. Over time, however, the capacity of NVM devices has increased faster than HDD capacity, and their price has dropped more quickly, so their use is increasing dramatically. In fact, SSDs and similar devices are now used in some laptop computers to make them smaller, faster, and more energy-efficient.

Because NVM devices can be much faster than hard disk drives, standard bus interfaces can cause a major limit on throughput. Some NVM devices are designed to connect directly to the system bus (PCIe, for example). This technology is changing other traditional aspects of computer design as well.

Figure 11.3 A 3.5-inch SSD circuit board.

Some systems use it as a direct replacement for disk drives, while others use it as a new cache tier, moving data among magnetic disks, NVM, and main memory to optimize performance.

NAND semiconductors have some characteristics that present their own storage and reliability challenges. For example, they can be read and written in a "page" increment (similar to a sector), but data cannot be overwritten—rather, the NAND cells have to be erased first. The erasure, which occurs in a "block" increment that is several pages in size, takes much more time than a read (the fastest operation) or a write (slower than read, but much faster than erase). Helping the situation is that NVM flash devices are composed of many die, with many datapaths to each die, so operations can happen in parallel (each using a datapath). NAND semiconductors also deteriorate with every erase cycle, and after approximately 100,000 program-erase cycles (the specific number varies depending on the medium), the cells no longer retain data. Because of the write wear, and because there are no moving parts, NAND NVM lifespan is not measured in years but in **Drive Writes Per Day** (**DWPD**). That measure is how many times the drive capacity can be written per day before the drive fails. For example, a 1 TB NAND drive with a 5 DWPD rating is expected to have 5 TB per day written to it for the warranty period without failure.

These limitations have led to several ameliorating algorithms. Fortunately, they are usually implemented in the NVM device controller and are not of concern to the operating system. The operating system simply reads and writes logical blocks, and the device manages how that is done. (Logical blocks are discussed in more detail in Section 11.1.5.) However, NVM devices have performance variations based on their operating algorithms, so a brief discussion of what the controller does is warranted.

11.1.2.2 NAND Flash Controller Algorithms

Because NAND semiconductors cannot be overwritten once written, there are usually pages containing invalid data. Consider a file-system block, written once and then later written again. If no erase has occurred in the meantime, the page first written has the old data, which are now invalid, and the second page has the current, good version of the block. A NAND block containing valid and invalid pages is shown in Figure 11.4. To track which logical blocks contain valid data, the controller maintains a **flas translation layer** (**FTL**). This table maps which physical pages contain currently valid logical blocks. It also

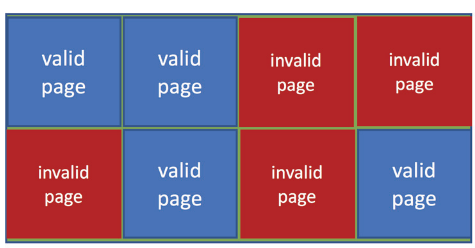


Figure 11.4 A NAND block with valid and invalid pages.

tracks physical block state—that is, which blocks contain only invalid pages and therefore can be erased.

Now consider a full SSD with a pending write request. Because the SSD is full, all pages have been written to, but there might be a block that contains no valid data. In that case, the write could wait for the erase to occur, and then the write could occur. But what if there are no free blocks? There still could be some space available if individual pages are holding invalid data. In that case, **garbage collection** could occur—good data could be copied to other locations, freeing up blocks that could be erased and could then receive the writes. However, where would the garbage collection store the good data? To solve this problem and improve write performance, the NVM device uses **over-provisioning**. The device sets aside a number of pages (frequently 20 percent of the total) as an area always available to write to. Blocks that are totally invalid by garbage collection, or write operations invalidating older versions of the data, are erased and placed in the over-provisioning space if the device is full or returned to the free pool.

The over-provisioning space can also help with **wear leveling**. If some blocks are erased repeatedly, while others are not, the frequently erased blocks will wear out faster than the others, and the entire device will have a shorter lifespan than it would if all the blocks wore out concurrently. The controller tries to avoid that by using various algorithms to place data on less-erased blocks so that subsequent erases will happen on those blocks rather than on the more erased blocks, leveling the wear across the entire device.

In terms of data protection, like HDDs, NVM devices provide error-correcting codes, which are calculated and stored along with the data during writing and read with the data to detect errors and correct them if possible. (Error-correcting codes are discussed in Section 11.5.1.) If a page frequently has correctible errors, the page might be marked as bad and not used in subsequent writes. Generally, a single NVM device, like an HDD, can have a catastrophic failure in which it corrupts or fails to reply to read or write requests. To allow data to be recoverable in those instances, RAID protection is used.

11.1.3 Volatile Memory

It might seem odd to discuss volatile memory in a chapter on mass-storage structure, but it is justifiable because DRAM is frequently used as a mass-storage device. Specifically, **RAM drives** (which are known by many names, including RAM disks) act like secondary storage but are created by device drivers that carve out a section of the system's DRAM and present it to the rest of the system as it if were a storage device. These "drives" can be used as raw block devices, but more commonly, file systems are created on them for standard file operations.

Computers already have buffering and caching, so what is the purpose of yet another use of DRAM for temporary data storage? After all, DRAM is volatile, and data on a RAM drive does not survive a system crash, shutdown, or power down. Caches and buffers are allocated by the programmer or operating system, whereas RAM drives allow the user (as well as the programmer) to place

MAGNETIC TAPES

Magnetic tape was used as an early secondary-storage medium. Although it is nonvolatile and can hold large quantities of data, its access time is slow compared with that of main memory and drives. In addition, random access to magnetic tape is about a thousand times slower than random access to HDDs and about a hundred thousand times slower than random access to SSDs so tapes are not very useful for secondary storage. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can read and write data at speeds comparable to HDDs. Tape capacities vary greatly, depending on the particular kind of tape drive, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-6 (Figure 11.5) and SDLT.

Figure 11.5 An LTO-6 Tape drive with tape cartridge inserted.

data in memory for temporary safekeeping using standard file operations. In fact, RAM drive functionality is useful enough that such drives are found in all major operating systems. On Linux there is `/dev/ram`, on macOS the `diskutil` command creates them, Windows has them via third-party tools, and Solaris and Linux create `/tmp` at boot time of type "tmpfs", which is a RAM drive.

RAM drives are useful as high-speed temporary storage space. Although NVM devices are fast, DRAM is much faster, and I/O operations to RAM drives are the fastest way to create, read, write, and delete files and their contents. Many programs use (or could benefit from using) RAM drives for storing temporary files. For example, programs can share data easily by writing and reading files from a RAM drive. For another example, Linux at boot time creates a temporary root file system (`initrd`) that allows other parts of the system to have access to a root file system and its contents before the parts of the operating system that understand storage devices are loaded.

11.1.4 Secondary Storage Connection Methods

A secondary storage device is attached to a computer by the system bus or an **I/O bus**. Several kinds of buses are available, including **advanced technology attachment** (**ATA**), **serial ATA** (**SATA**), **eSATA**, **serial attached SCSI** (**SAS**), **universal serial bus** (**USB**), and **fibr channel** (**FC**). The most common connection method is SATA. Because NVM devices are much faster than HDDs, the industry created a special, fast interface for NVM devices called **NVM express** (**NVMe**). NVMe directly connects the device to the system PCI bus, increasing throughput and decreasing latency compared with other connection methods.

The data transfers on a bus are carried out by special electronic processors called **controllers** (or **host-bus adapters** (**HBA**)). The **host controller** is the controller at the computer end of the bus. A **device controller** is built into each storage device. To perform a mass storage I/O operation, the computer places a command into the host controller, typically using memory-mapped I/O ports, as described in Section 12.2.1. The host controller then sends the command via messages to the device controller, and the controller operates the drive hardware to carry out the command. Device controllers usually have a built-in cache. Data transfer at the drive happens between the cache and the storage media, and data transfer to the host, at fast electronic speeds, occurs between the cache host DRAM via DMA.

11.1.5 Address Mapping

Storage devices are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer. Each logical block maps to a physical sector or semiconductor page. The one-dimensional array of logical blocks is mapped onto the sectors or pages of the device. Sector 0 could be the first sector of the first track on the outermost cylinder on an HDD, for example. The mapping proceeds in order through that track, then through the rest of the tracks on that cylinder, and then through the rest of the cylinders, from outermost to innermost. For NVM the mapping is from a tuple (finite ordered list) of chip, block, and page to an array of logical blocks. A logical block address (**LBA**) is easier for algorithms to use than a sector, cylinder, head tuple or chip, block, page tuple.

By using this mapping on an HDD, we can—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform this translation, for three reasons. First, most drives have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the drive. The logical block address stays sequential, but the physical sector location changes. Second, the number of sectors per track is not a constant on some drives. Third, disk manufacturers manage LBA to physical address mapping internally, so in current drives there is little relationship between LBA and physical sectors. In spite of these physical address vagaries, algorithms that deal with HDDs tend to assume that logical addresses are relatively related to physical addresses. That is, ascending logical addresses tend to mean ascending physical address.

Let's look more closely at the second reason. On media that use **constant linear velocity** (**CLV**), the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant; in this case, the density of bits decreases from inner tracks to outer tracks to keep the data rate constant (and performance relatively the same no matter where data is on the drive). This method is used in hard disks and is known as **constant angular velocity** (**CAV**).

The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.

Note that there are more types of storage devices than are reasonable to cover in an operating systems text. For example, there are "shingled magnetic recording" hard drives with higher density but worse performance than mainstream HDDs (see http://www.tomsitpro.com/articles/shingled-magnetic-recoding-smr-101-basics,2-933.html). There are also combination devices that include NVM and HDD technology, or volume managers (see Section 11.5) that can knit together NVM and HDD devices into a storage unit faster than HDD but lower cost than NVM. These devices have different characteristics from the more common devices, and might need different caching and scheduling algorithms to maximize performance.

11.2 HDD Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For HDDs, meeting this responsibility entails minimizing access time and maximizing data transfer bandwidth.

For HDDs and other mechanical storage devices that use platters, access time has two major components, as mentioned in Section 11.1. The seek time is the time for the device arm to move the heads to the cylinder containing the desired sector, and the rotational latency is the additional time for the platter to rotate the desired sector to the head. The device **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by managing the order in which storage I/O requests are serviced.

Whenever a process needs I/O to or from the drive, it issues a system call to the operating system. The request specifies several pieces of information:

- Whether this operation is input or output
- The open file handle indicating the file to operate on
- What the memory address for the transfer is
- The amount of data to transfer

If the desired drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive. For a multiprogramming system with many processes, the device queue may often have several pending requests.

The existence of a queue of requests to a device that can have its performance optimized by avoiding head seeks allows device drivers a chance to improve performance via queue ordering.

In the past, HDD interfaces required that the host specify which track and which head to use, and much effort was spent on disk scheduling algorithms. Drives newer than the turn of the century not only do not expose these controls to the host, but also map LBA to physical addresses under drive control. The current goals of disk scheduling include fairness, timeliness, and optimizations, such as bunching reads or writes that appear in sequence, as drives perform best with sequential I/O. Therefore some scheduling effort is still useful. Any one of several disk-scheduling algorithms can be used, and we discuss them next. Note that absolute knowledge of head location and physical block/cylinder locations is generally not possible on modern drives. But as a rough approximation, algorithms can assume that increasing LBAs mean increasing physical addresses, and LBAs close together equate to physical block proximity.

11.2.1 FCFS Scheduling

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm (or FIFO). This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders

98, 183, 37, 122, 14, 124, 65, 67,

in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure 11.6.

The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

11.2.2 SCAN Scheduling

In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the **elevator algorithm**, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

Let's return to our example to illustrate. Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know

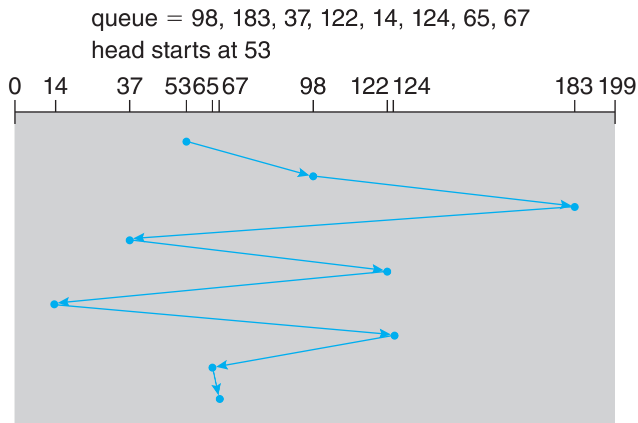


Figure 11.6 FCFS disk scheduling.

the direction of head movement in addition to the head's current position. Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 11.7). If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

Assuming a uniform distribution of requests for cylinders, consider the density of requests when the head reaches one end and reverses direction. At this point, relatively few requests are immediately in front of the head, since these cylinders have recently been serviced. The heaviest density of requests is at the other end of the disk. These requests have also waited the longest, so why not go there first? That is the idea of the next algorithm.

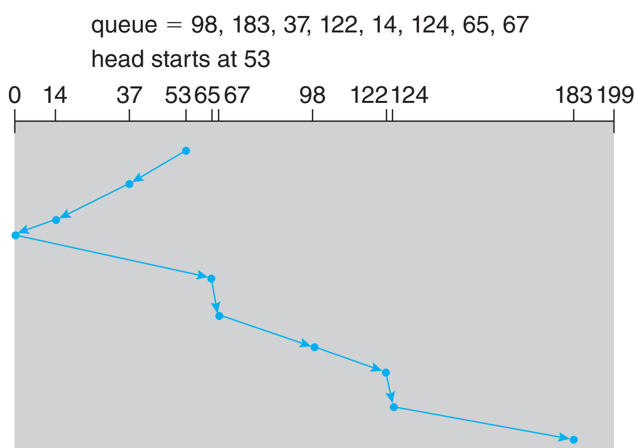


Figure 11.7 SCAN disk scheduling.

11.2.3 C-SCAN Scheduling

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.

Let's return to our example to illustrate. Before applying C-SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in which the requests are scheduled. Assuming that the requests are scheduled when the disk arm is moving from 0 to 199 and that the initial head position is again 53, the request will be served as depicted in Figure 11.8. The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

11.2.4 Selection of a Disk-Scheduling Algorithm

There are many disk-scheduling algorithms not included in this coverage, because they are rarely used. But how do operating system designers decide which to implement, and deployers chose the best to use? For any particular list of requests, we can define an optimal order of retrieval, but the computation needed to find an optimal schedule may not justify the savings over SCAN. With any scheduling algorithm, however, performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request. Then, all scheduling algorithms behave the same, because they have only one choice of where to move the disk head: they all behave like FCFS scheduling.

SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem. There can still be starvation though, which drove Linux to create the **deadline** scheduler. This scheduler maintains separate read and write queues, and gives reads priority because processes are more likely to block on read than write. The queues are

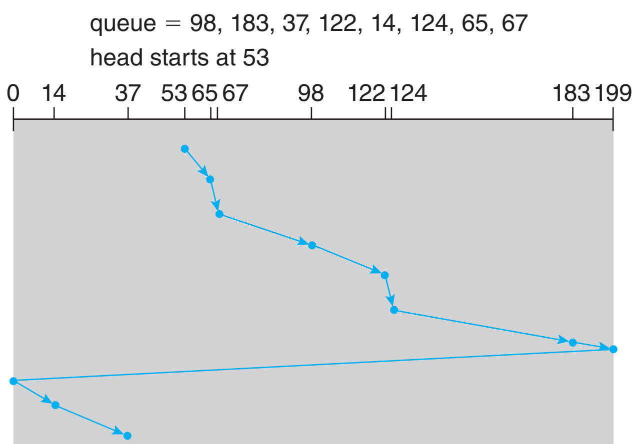


Figure 11.8 C-SCAN disk scheduling.

sorted in LBA order, essentially implementing C-SCAN. All I/O requests are sent in a batch in this LBA order. Deadline keeps four queues: two read and two write, one sorted by LBA and the other by FCFS. It checks after each batch to see if there are requests in the FCFS queues older than a configured age (by default, 500 ms). If so, the LBA queue (read or write) containing that request is selected for the next batch of I/O.

The deadline I/O scheduler is the default in the Linux RedHat 7 distribution, but **RHEL** 7 also includes two others. NOOP is preferred for CPU-bound systems using fast storage such as NVM devices, and the **Completely Fair Queueing scheduler** (**CFQ**) is the default for SATA drives. CFQ maintains three queues (with insertion sort to keep them sorted in LBA order): real time, best effort (the default), and idle. Each has exclusive priority over the others, in that order, with starvation possible. It uses historical data, anticipating if a process will likely issue more I/O requests soon. If it so determines, it idles waiting for the new I/O, ignoring other queued requests. This is to minimize seek time, assuming locality of reference of storage I/O requests, per process. Details of these schedulers can be found in https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Performance_Tuning_Guide/index.html.

11.3 NVM Scheduling

The disk-scheduling algorithms just discussed apply to mechanical platter-based storage like HDDs. They focus primarily on minimizing the amount of disk head movement. NVM devices do not contain moving disk heads and commonly use a simple FCFS policy. For example, the Linux **NOOP** scheduler uses an FCFS policy but modifies it to merge adjacent requests. The observed behavior of NVM devices indicates that the time required to service reads is uniform but that, because of the properties of flash memory, write service time is not uniform. Some SSD schedulers have exploited this property and merge only adjacent write requests, servicing all read requests in FCFS order.

As we have seen, I/O can occur sequentially or randomly. Sequential access is optimal for mechanical devices like HDD and tape because the data to be read or written is near the read/write head. Random-access I/O, which is measured in **input/output operations per second** (**IOPS**), causes HDD disk head movement. Naturally, random access I/O is much faster on NVM. An HDD can produce hundreds of IOPS, while an SSD can produce hundreds of thousands of IOPS.

NVM devices offer much less of an advantage for raw sequential throughput, where HDD head seeks are minimized and reading and writing of data to the media are emphasized. In those cases, for reads, performance for the two types of devices can range from equivalent to an order of magnitude advantage for NVM devices. Writing to NVM is slower than reading, decreasing the advantage. Furthermore, while write performance for HDDs is consistent throughout the life of the device, write performance for NVM devices varies depending on how full the device is (recall the need for garbage collection and over-provisioning) and how “worn” it is. An NVM device near its end of life due to many erase cycles generally has much worse performance than a new device.

One way to improve the lifespan and performance of NVM devices over time is to have the file system inform the device when files are deleted, so that the device can erase the blocks those files were stored on. This approach is discussed further in Section 14.5.6.

Let's look more closely at the impact of garbage collection on performance. Consider an NVM device under random read and write load. Assume that all blocks have been written to, but there is free space available. Garbage collection must occur to reclaim space taken by invalid data. That means that a write might cause a read of one or more pages, a write of the good data in those pages to overprovisioning space, an erase of the all-invalid-data block, and the placement of that block into overprovisioning space. In summary, one write request eventually causes a page write (the data), one or more page reads (by garbage collection), and one or more page writes (of good data from the garbage-collected blocks). The creation of I/O requests not by applications but by the NVM device doing garbage collection and space management is called **write amplificatio** and can greatly impact the write performance of the device. In the worst case, several extra I/Os are triggered with each write request.

11.4 Error Detection and Correction

Error detection and correction are fundamental to many areas of computing, including memory, networking, and storage. **Error detection** determines if a problem has occurred — for example a bit in DRAM spontaneously changed from a 0 to a 1, the contents of a network packet changed during transmission, or a block of data changed between when it was written and when it was read. By detecting the issue, the system can halt an operation before the error is propagated, report the error to the user or administrator, or warn of a device that might be starting to fail or has already failed.

Memory systems have long detected certain errors by using parity bits. In this scenario, each byte in a memory system has a parity bit associated with it that records whether the number of bits in the byte set to 1 is even (parity = 0) or odd (parity = 1). If one of the bits in the byte is damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus does not match the stored parity. Similarly, if the stored parity bit is damaged, it does not match the computed parity. Thus, all single-bit errors are detected by the memory system. A double-bit-error might go undetected, however. Note that parity is easily calculated by performing an XOR (for "eXclusive OR") of the bits. Also note that for every byte of memory, we now need an extra bit of memory to store the parity.

Parity is one form of **checksums**, which use modular arithmetic to compute, store, and compare values on fixed-length words. Another error-detection method, common in networking, is a **cyclic redundancy check** (**CRCs**), which uses a hash function to detect multiple-bit errors (see http://www.mathpages.com/home/kmath458/kmath458.htm).

An **error-correction code** (**ECC**) not only detects the problem, but also corrects it. The correction is done by using algorithms and extra amounts of storage. The codes vary based on how much extra storage they need and how many errors they can correct. For example, disks drives use per-sector ECC and flash drives per-page ECC. When the controller writes a sector/page of data during normal I/O, the ECC is written with a value calculated from all the bytes in the data being written. When the sector/page is read, the ECC is recalculated and compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data have become corrupted and that the storage media may be bad (Section 11.5.3). The ECC is error correcting because it contains enough information, if only a few bits of data have been corrupted, to enable the controller to identify which bits have changed and calculate what their correct values should be. It then reports a recoverable **soft error**. If too many changes occur, and the ECC cannot correct the error, a noncorrectable **hard error** is signaled. The controller automatically does the ECC processing whenever a sector or page is read or written.

Error detection and correction are frequently differentiators between consumer products and enterprise products. ECC is used in some systems for DRAM error correction and data path protection, for example.

11.5 Storage Device Management

The operating system is responsible for several other aspects of storage device management, too. Here, we discuss drive initialization, booting from a drive, and bad-block recovery.

11.5.1 Drive Formatting, Partitions, and Volumes

A new storage device is a blank slate: it is just a platter of a magnetic recording material or a set of uninitialized semiconductor storage cells. Before a storage device can store data, it must be divided into sectors that the controller can read and write. NVM pages must be initialized and the FTL created. This process is called **low-level formatting**, or **physical formatting**. Low-level formatting fills the device with a special data structure for each storage location. The data structure for a sector or page typically consists of a header, a data area, and a trailer. The header and trailer contain information used by the controller, such as a sector/page number and an error detection or correction code.

Most drives are low-level-formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the device and to initialize the mapping from logical block numbers to defect-free sectors or pages on the media. It is usually possible to choose among a few sector sizes, such as 512 bytes and 4KB. Formatting a disk with a larger sector size means that fewer sectors can fit on each track, but it also means that fewer headers and trailers are written on each track and more space is available for user data. Some operating systems can handle only one specific sector size.

Before it can use a drive to hold files, the operating system still needs to record its own data structures on the device. It does so in three steps.

The first step is to **partition** the device into one or more groups of blocks or pages. The operating system can treat each partition as though it were a separate device. For instance, one partition can hold a file system containing a copy of the operating system's executable code, another the swap space, and another a file system containing the user files. Some operating systems and file systems perform the partitioning automatically when an entire device is to be managed by the file system. The partition information is written in a fixed format at a fixed location on the storage device. In Linux, the `fdisk` command is used to manage partitions on storage devices. The device, when recognized by the operating system, has its partition information read, and the operating system then creates device entries for the partitions (in `/dev` in Linux). From there, a configuration file, such as `/etc/fstab`, tells the operating system to mount each partition containing a file system at a specified location and to use mount options such as read-only. **Mounting** a file system is making the file system available for use by the system and its users.

The second step is volume creation and management. Sometimes, this step is implicit, as when a file system is placed directly within a partition. That **volume** is then ready to be mounted and used. At other times, volume creation and management is explicit—for example when multiple partitions or devices will be used together as a RAID set (see Section 11.8) with one or more file systems spread across the devices. The Linux volume manager `lvm2` can provide these features, as can commercial third-party tools for Linux and other operating systems. ZFS provides both volume management and a file system integrated into one set of commands and features. (Note that "volume" can also mean any mountable file system, even a file containing a file system such as a CD image.)

The third step is **logical formatting**, or creation of a file system. In this step, the operating system stores the initial file-system data structures onto the device. These data structures may include maps of free and allocated space and an initial empty directory.

The partition information also indicates if a partition contains a bootable file system (containing the operating system). The partition labeled for boot is used to establish the root of the file system. Once it is mounted, device links for all other devices and their partitions can be created. Generally, a computer's "file system" consists of all mounted volumes. On Windows, these are separately named via a letter (C:, D:, E:). On other systems, such as Linux, at boot time the boot file system is mounted, and other file systems can be mounted within that tree structure (as discussed in Section 13.3). On Windows, the file system interface makes it clear when a given device is being used, while in Linux a single file access might traverse many devices before the requested file in the requested file system (within a volume) is accessed. Figure 11.9 shows the Windows 7 `Disk Management` tool displaying three volumes (C:, E:, and F:). Note that E: and F: are each in a partition of the "Disk 1" device and that there is unallocated space on that device for more partitions (possibly containing file systems).

To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters**. Device I/O is done via blocks, but file system I/O is done via clusters, effectively assuring that I/O has more sequential-access and fewer random-access characteristics. File systems try to group file contents near its metadata as well, reducing HDD head seeks when operating on a file, for example.

Some operating systems give special programs the ability to use a partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the **raw disk**, and I/O to this array is termed ***raw I/O***. It can be used for swap space (see Section 11.6.2), for example, and some database systems prefer raw I/O because it enables them to control

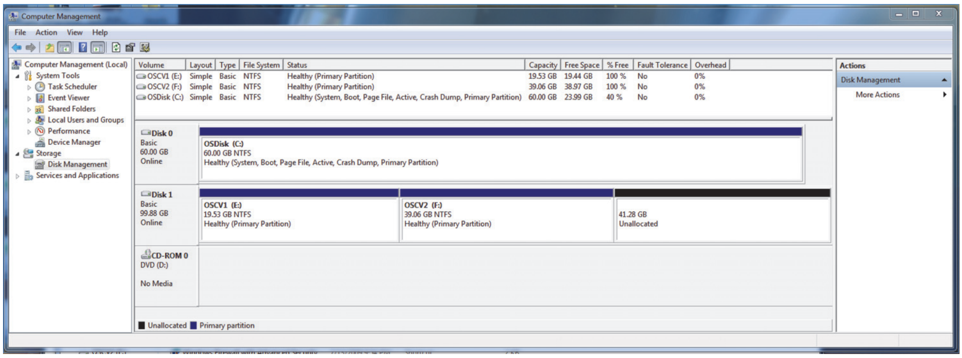


Figure 11.9 Windows 7 `Disk Management` tool showing devices, partitions, volumes, and file systems.

the exact location where each database record is stored. Raw I/O bypasses all the file-system services, such as the buffer cache, file locking, prefetching, space allocation, file names, and directories. We can make certain applications more efficient by allowing them to implement their own special-purpose storage services on a raw partition, but most applications use a provided file system rather than managing data themselves. Note that Linux generally does not support raw I/O but can achieve similar access by using the DIRECT flag to the `open()` system call.

11.5.2 Boot Block

For a computer to start running—for instance, when it is powered up or rebooted—it must have an initial program to run. This initial **bootstrap** loader tends to be simple. For most computers, the bootstrap is stored in NVM flash memory firmware on the system motherboard and mapped to a known memory location. It can be updated by product manufacturers as needed, but also can be written to by viruses, infecting the system. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory.

This tiny bootstrap loader program is also smart enough to bring in a full bootstrap program from secondary storage. The full bootstrap program is stored in the "boot blocks" at a fixed location on the device. The default Linux bootstrap loader is grub2 (https://www.gnu.org/software/grub/manual/grub.html/). A device that has a boot partition is called a **boot disk** or **system disk**.

The code in the bootstrap NVM instructs the storage controller to read the boot blocks into memory (no device drivers are loaded at this point) and then starts executing that code. The full bootstrap program is more sophisticated than the bootstrap loader: it is able to load the entire operating system from a non-fixed location on the device and to start the operating system running.

Let's consider as an example the boot process in Windows. First, note that Windows allows a drive to be divided into partitions, and one partition—identified as the **boot partition**—contains the operating system and device drivers. The Windows system places its boot code in the first logical block on the hard disk or first page of the NVM device, which it terms the **master boot**

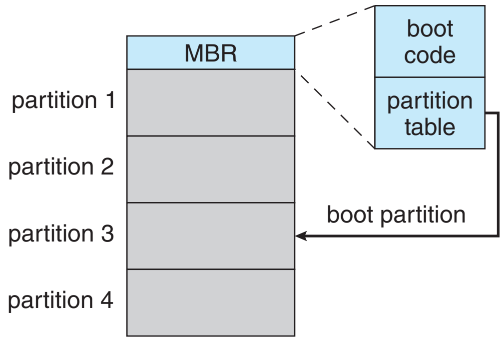


Figure 11.10 Booting from a storage device in Windows.

record, or **MBR**. Booting begins by running code that is resident in the system's firmware. This code directs the system to read the boot code from the MBR, understanding just enough about the storage controller and storage device to load a sector from it. In addition to containing boot code, the MBR contains a table listing the partitions for the drive and a flag indicating which partition the system is to be booted from, as illustrated in Figure 11.10. Once the system identifies the boot partition, it reads the first sector/page from that partition (called the **boot sector**), which directs it to the kernel. It then continues with the remainder of the boot process, which includes loading the various subsystems and system services.

11.5.3 Bad Blocks

Because disks have moving parts and small tolerances (recall that the disk head flies just above the disk surface), they are prone to failure. Sometimes the failure is complete; in this case, the disk needs to be replaced and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective. Most disks even come from the factory with **bad blocks**. Depending on the disk and controller in use, these blocks are handled in a variety of ways.

On older disks, such as some disks with IDE controllers, bad blocks are handled manually. One strategy is to scan the disk to find bad blocks while the disk is being formatted. Any bad blocks that are discovered are flagged as unusable so that the file system does not allocate them. If blocks go bad during normal operation, a special program (such as the Linux `badblocks` command) must be run manually to search for the bad blocks and to lock them away. Data that resided on the bad blocks usually are lost.

More sophisticated disks are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level formatting at the factory and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding**.

A typical bad-sector transaction might be as follows:

- The operating system tries to read logical block 87.

- The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system as an I/O error.
- The device controller replaces the bad sector with a spare.
- After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Note that such a redirection by the controller could invalidate any optimization by the operating system's disk-scheduling algorithm! For this reason, most disks are formatted to provide a few spare sectors in each cylinder and a spare cylinder as well. When a bad block is remapped, the controller uses a spare sector from the same cylinder, if possible.

As an alternative to sector sparing, some controllers can be instructed to replace a bad block by **sector slipping**. Here is an example: Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Sector slipping then remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18 so that sector 17 can be mapped to it.

Recoverable soft errors may trigger a device activity in which a copy of the block data is made and the block is spared or slipped. An unrecoverable **hard error**, however, results in lost data. Whatever file was using that block must be repaired (for instance, by restoration from a backup tape), and that requires manual intervention.

NVM devices also have bits, bytes, and even pages that either are nonfunctional at manufacturing time or go bad over time. Management of those faulty areas is simpler than for HDDs because there is no seek time performance loss to be avoided. Either multiple pages can be set aside and used as replacement locations, or space from the over-provisioning area can be used (decreasing the usable capacity of the over-provisioning area). Either way, the controller maintains a table of bad pages and never sets those pages as available to write to, so they are never accessed.

11.6 Swap-Space Management

Swapping was first presented in Section 9.5, where we discussed moving entire processes between secondary storage and main memory. Swapping in that setting occurs when the amount of physical memory reaches a critically low point and processes are moved from memory to swap space to free available memory. In practice, very few modern operating systems implement swapping in this fashion. Rather, systems now combine swapping with virtual memory techniques (Chapter 10) and swap pages, not necessarily entire processes. In fact, some systems now use the terms "swapping" and "paging" interchangeably, reflecting the merging of these two concepts.

Swap-space management is another low-level task of the operating system. Virtual memory uses secondary storage space as an extension of main memory. Since drive access is much slower than memory access, using swap space significantly decreases system performance. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system. In this section, we discuss how swap space is used, where swap space is located on storage devices, and how swap space is managed.

11.6.1 Swap-Space Use

Swap space is used in various ways by different operating systems, depending on the memory-management algorithms in use. For instance, systems that implement swapping may use swap space to hold an entire process image, including the code and data segments. Paging systems may simply store pages that have been pushed out of main memory. The amount of swap space needed on a system can therefore vary from a few megabytes of disk space to gigabytes, depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used.

Note that it may be safer to overestimate than to underestimate the amount of swap space required, because if a system runs out of swap space it may be forced to abort processes or may crash entirely. Overestimation wastes secondary storage space that could otherwise be used for files, but it does no other harm. Some systems recommend the amount to be set aside for swap space. Solaris, for example, suggests setting swap space equal to the amount by which virtual memory exceeds pageable physical memory. In the past, Linux has suggested setting swap space to double the amount of physical memory. Today, the paging algorithms have changed, and most Linux systems use considerably less swap space.

Some operating systems—including Linux—allow the use of multiple swap spaces, including both files and dedicated swap partitions. These swap spaces are usually placed on separate storage devices so that the load placed on the I/O system by paging and swapping can be spread over the system's I/O bandwidth.

11.6.2 Swap-Space Location

A swap space can reside in one of two places: it can be carved out of the normal file system, or it can be in a separate partition. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space.

Alternatively, swap space can be created in a separate **raw partition**. No file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition. This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems, when it is used (recall that swap space is used for swapping and paging). Internal fragmentation may increase, but this trade-off is acceptable because the life of data in the swap space generally is much shorter than that of files in the file system. Since swap space is reinitialized at boot time, any fragmentation is short-lived. The raw-partition approach creates a fixed amount of swap space during disk partitioning. Adding more swap space requires either repartitioning the device (which involves moving the other file-system partitions or destroying them and restoring them from backup) or adding another swap space elsewhere.

Some operating systems are flexible and can swap both in raw partitions and in file-system space. Linux is an example: the policy and implementation are separate, allowing the machine's administrator to decide which type of swapping to use. The trade-off is between the convenience of allocation and management in the file system and the performance of swapping in raw partitions.

11.6.3 Swap-Space Management: An Example

We can illustrate how swap space is used by following the evolution of swapping and paging in various UNIX systems. The traditional UNIX kernel started with an implementation of swapping that copied entire processes between contiguous disk regions and memory. UNIX later evolved to a combination of swapping and paging as paging hardware became available.

In Solaris 1 (SunOS), the designers changed standard UNIX methods to improve efficiency and reflect technological developments. When a process executes, text-segment pages containing code are brought in from the file system, accessed in main memory, and thrown away if selected for pageout. It is more efficient to reread a page from the file system than to write it to swap space and then reread it from there. Swap space is only used as a backing store for pages of **anonymous** memory (memory not backed by any file), which includes memory allocated for the stack, heap, and uninitialized data of a process.

More changes were made in later versions of Solaris. The biggest change is that Solaris now allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created. This scheme gives better performance on modern computers, which have more physical memory than older systems and tend to page less.

Linux is similar to Solaris in that swap space is now used only for anonymous memory. Linux allows one or more swap areas to be established. A swap area may be in either a swap file on a regular file system or a dedicated swap partition. Each swap area consists of a series of 4-KB **page slots**, which are used to hold swapped pages. Associated with each swap area is a **swap map**—an array of integer counters, each corresponding to a page slot in the swap area. If the value of a counter is 0, the corresponding page slot is available. Values greater than 0 indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page. For example, a value of 3 indicates that the swapped page is mapped to three different processes (which can occur if the swapped page is storing a region of memory shared by three processes). The data structures for swapping on Linux systems are shown in Figure 11.11.

11.7 Storage Attachment

Computers access secondary storage in three ways: via host-attached storage, network-attached storage, and cloud storage.

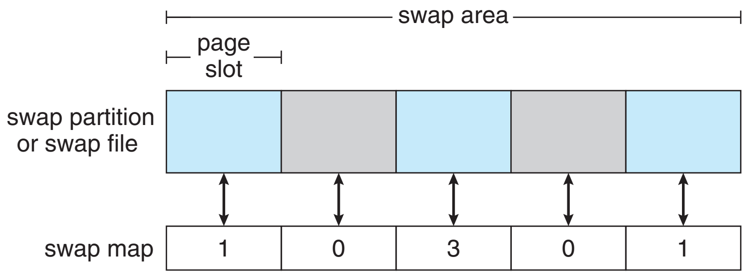


Figure 11.11 The data structures for swapping on Linux systems.

11.7.1 Host-Attached Storage

Host-attached storage is storage accessed through local I/O ports. These ports use several technologies, the most common being SATA, as mentioned earlier. A typical system has one or a few SATA ports.

To allow a system to gain access to more storage, either an individual storage device, a device in a chassis, or multiple drives in a chassis can be connected via USB FireWire or Thunderbolt ports and cables.

High-end workstations and servers generally need more storage or need to share storage, so use more sophisticated I/O architectures, such as **fibr channel** (**FC**), a high-speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. Because of the large address space and the switched nature of the communication, multiple hosts and storage devices can attach to the fabric, allowing great flexibility in I/O communication.

A wide variety of storage devices are suitable for use as host-attached storage. Among these are HDDs; NVM devices; CD, DVD, Blu-ray, and tape drives; and storage-area networks (**SANs**) (discussed in Section 11.7.4). The I/O commands that initiate data transfers to a host-attached storage device are reads and writes of logical data blocks directed to specifically identified storage units (such as bus ID or target logical unit).

11.7.2 Network-Attached Storage

Network-attached storage (**NAS**) (Figure 11.12) provides access to storage across a network. An NAS device can be either a special-purpose storage system or a general computer system that provides its storage to other hosts across the network. Clients access network-attached storage via a remote-procedure-call interface such as NFS for UNIX and Linux systems or CIFS for Windows machines. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network—usually the same local-area network (LAN) that carries all data traffic to the clients. The network-attached storage unit is usually implemented as a storage array with software that implements the RPC interface.

CIFS and NFS provide various locking features, allowing the sharing of files between hosts accessing a NAS with those protocols. For example, a user logged in to multiple NAS clients can access her home directory from all of those clients, simultaneously.

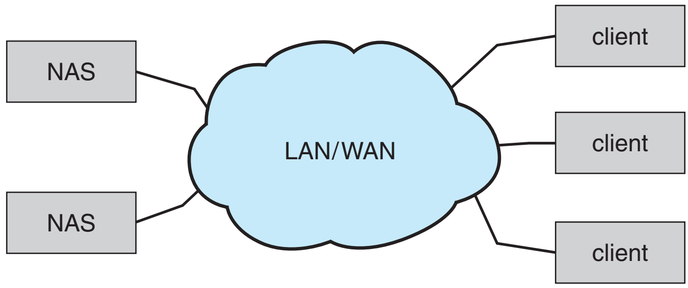


Figure 11.12 Network-attached storage.

Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage with the same ease of naming and access enjoyed with local host-attached storage. However, it tends to be less efficient and have lower performance than some direct-attached storage options.

iSCSI is the latest network-attached storage protocol. In essence, it uses the IP network protocol to carry the SCSI protocol. Thus, networks—rather than SCSI cables—can be used as the interconnects between hosts and their storage. As a result, hosts can treat their storage as if it were directly attached, even if the storage is distant from the host. Whereas NFS and CIFS present a file system and send parts of files across the network, iSCSI sends logical blocks across the network and leaves it to the client to use the blocks directly or create a file system with them.

11.7.3 Cloud Storage

Section 1.10.5 discussed cloud computing. One offering from cloud providers is **cloud storage**. Similar to network-attached storage, cloud storage provides access to storage across a network. Unlike NAS, the storage is accessed over the Internet or another WAN to a remote data center that provides storage for a fee (or even for free).

Another difference between NAS and cloud storage is how the storage is accessed and presented to users. NAS is accessed as just another file system if the CIFS or NFS protocols are used, or as a raw block device if the iSCSI protocol is used. Most operating systems have these protocols integrated and present NAS storage in the same way as other storage. In contrast, cloud storage is API based, and programs use the APIs to access the storage. Amazon S3 is a leading cloud storage offering. Dropbox is an example of a company that provides apps to connect to the cloud storage that it provides. Other examples include Microsoft OneDrive and Apple iCloud.

One reason that APIs are used instead of existing protocols is the latency and failure scenarios of a WAN. NAS protocols were designed for use in LANs, which have lower latency than WANs and are much less likely to lose connectivity between the storage user and the storage device. If a LAN connection fails, a system using NFS or CIFS might hang until it recovers. With cloud storage, failures like that are more likely, so an application simply pauses access until connectivity is restored.

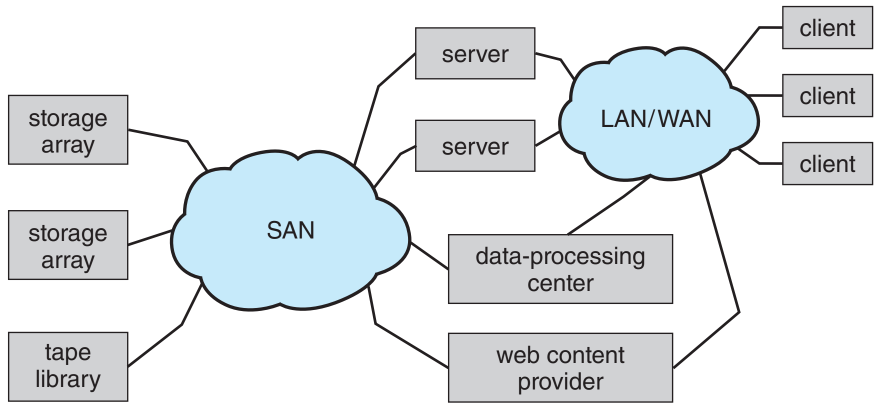


Figure 11.13 Storage-area network.

11.7.4 Storage-Area Networks and Storage Arrays

One drawback of network-attached storage systems is that the storage I/O operations consume bandwidth on the data network, thereby increasing the latency of network communication. This problem can be particularly acute in large client–server installations—the communication between servers and clients competes for bandwidth with the communication among servers and storage devices.

A **storage-area network** (**SAN**) is a private network (using storage protocols rather than networking protocols) connecting servers and storage units, as shown in Figure 11.13. The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts. The storage arrays can be RAID protected or unprotected drives (**Just a Bunch of Disks** (**JBOD**)). A SAN switch allows or prohibits access between the hosts and the storage. As one example, if a host is running low on disk space, the SAN can be configured to allocate more storage to that host. SANs make it possible for clusters of servers to share the same storage and for storage arrays to include multiple direct host connections. SANs typically have more ports—and cost more—than storage arrays. SAN connectivity is over short distances and typically has no routing, so a NAS can have many more connected hosts than a SAN.

A storage array is a purpose-built device (see Figure 11.14) that includes SAN ports, network ports, or both. It also contains drives to store data and a controller (or redundant set of controllers) to manage the storage and allow access to the storage across the networks. The controllers are composed of CPUs, memory, and software that implement the features of the array, which can include network protocols, user interfaces, RAID protection, snapshots, replication, compression, deduplication, and encryption. Some of those functions are discussed in Chapter 14.

Some storage arrays include SSDs. An array may contain only SSDs, resulting in maximum performance but smaller capacity, or may include a mix of SSDs and HDDs, with the array software (or the administrator) selecting the best medium for a given use or using the SSDs as a cache and HDDs as bulk storage.

Figure 11.14 A storage array.

FC is the most common SAN interconnect, although the simplicity of iSCSI is increasing its use. Another SAN interconnect is **InfiniBan** (**IB**)—a special-purpose bus architecture that provides hardware and software support for high-speed interconnection networks for servers and storage units.

11.8 RAID Structure

Storage devices have continued to get smaller and cheaper, so it is now economically feasible to attach many drives to a computer system. Having a large number of drives in a system presents opportunities for improving the rate at which data can be read or written, if the drives are operated in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple drives. Thus, failure of one drive does not lead to loss of data. A variety of disk-organization techniques, collectively called **redundant arrays of independent disks** (**RAIDs**), are commonly used to address the performance and reliability issues.

In the past, RAIDs composed of small, cheap disks were viewed as a cost-effective alternative to large, expensive disks. Today, RAIDs are used for their higher reliability and higher data-transfer rate rather than for economic reasons. Hence, the ***I*** in ***RAID,*** which once stood for “inexpensive,” now stands for “independent.”

11.8.1 Improvement of Reliability via Redundancy

Let’s first consider the reliability of a RAID of HDDs. The chance that some disk out of a set of *N* disks will fail is much greater than the chance that a specific single disk will fail. Suppose that the **mean time between failures** (**MTBF**) of a single disk is 100,000 hours. Then the MTBF of some disk in an array of 100

STRUCTURING RAID

RAID storage can be structured in a variety of ways. For example, a system can have drives directly attached to its buses. In this case, the operating system or system software can implement RAID functionality. Alternatively, an intelligent host controller can control multiple attached devices and can implement RAID on those devices in hardware. Finally, a storage array can be used. A storage array, as just discussed, is a standalone unit with its own controller, cache, and drives. It is attached to the host via one or more standard controllers (for example, FC). This common setup allows an operating system or software without RAID functionality to have RAID-protected storage.

disks will be 100,000/100 = 1,000 hours, or 41.66 days, which is not long at all! If we store only one copy of the data, then each disk failure will result in loss of a significant amount of data—and such a high rate of data loss is unacceptable.

The solution to the problem of reliability is to introduce **redundancy**; we store extra information that is not normally needed but can be used in the event of disk failure to rebuild the lost information. Thus, even if a disk fails, data are not lost. RAID can be applied to NVM devices as well, although NVM devices have no moving parts and therefore are less likely to fail than HDDs.

The simplest (but most expensive) approach to introducing redundancy is to duplicate every drive. This technique is called **mirroring**. With mirroring, a logical disk consists of two physical drives, and every write is carried out on both drives. The result is called a **mirrored volume**. If one of the drives in the volume fails, the data can be read from the other. Data will be lost only if the second drive fails before the first failed drive is replaced.

The MTBF of a mirrored volume—where failure is the loss of data—depends on two factors. One is the MTBF of the individual drives. The other is the **mean time to repair**, which is the time it takes (on average) to replace a failed drive and to restore the data on it. Suppose that the failures of the two drives are independent; that is, the failure of one is not connected to the failure of the other. Then, if the MTBF of a single drive is 100,000 hours and the mean time to repair is 10 hours, the **mean time to data loss** of a mirrored drive system is $100,000^2/(2 * 10) = 500 * 10^6$ hours, or 57,000 years!

You should be aware that we cannot really assume that drive failures will be independent. Power failures and natural disasters, such as earthquakes, fires, and floods, may result in damage to both drives at the same time. Also, manufacturing defects in a batch of drives can cause correlated failures. As drives age, the probability of failure grows, increasing the chance that a second drive will fail while the first is being repaired. In spite of all these considerations, however, mirrored-drive systems offer much higher reliability than do single-drive systems.

Power failures are a particular source of concern, since they occur far more frequently than do natural disasters. Even with mirroring of drives, if writes are in progress to the same block in both drives, and power fails before both blocks are fully written, the two blocks can be in an inconsistent state. One solution to this problem is to write one copy first, then the next. Another is to add a solid-state nonvolatile cache to the RAID array. This write-back cache is protected from data loss during power failures, so the write can be considered complete at that point, assuming the cache has some kind of error protection and correction, such as ECC or mirroring.

11.8.2 Improvement in Performance via Parallelism

Now let's consider how parallel access to multiple drives improves performance. With mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either drive (as long as both in a pair are functional, as is almost always the case). The transfer rate of each read is the same as in a single-drive system, but the number of reads per unit time has doubled.

With multiple drives, we can improve the transfer rate as well (or instead) by striping data across the drives. In its simplest form, **data striping** consists of splitting the bits of each byte across multiple drives; such striping is called **bit-level striping**. For example, if we have an array of eight drives, we write bit i of each byte to drive i. The array of eight drives can be treated as a single drive with sectors that are eight times the normal size and, more important, have eight times the access rate. Every drive participates in every access (read or write); so the number of accesses that can be processed per second is about the same as on a single drive, but each access can read eight times as many data in the same time as on a single drive.

Bit-level striping can be generalized to include a number of drives that either is a multiple of 8 or divides 8. For example, if we use an array of four drives, bits i and $4+i$ of each byte go to drive i. Further, striping need not occur at the bit level. In **block-level striping**, for instance, blocks of a file are striped across multiple drives; with n drives, block i of a file goes to drive $(i \bmod n)+1$. Other levels of striping, such as bytes of a sector or sectors of a block, also are possible. Block-level striping is the only commonly available striping.

Parallelism in a storage system, as achieved through striping, has two main goals:

1. Increase the throughput of multiple small accesses (that is, page accesses) by load balancing.
2. Reduce the response time of large accesses.

11.8.3 RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability. Numerous schemes to provide redundancy at lower cost by using disk striping combined with "parity" bits (which we describe shortly) have been proposed. These schemes have different cost–performance trade-offs and are classified according to levels called **RAID levels**. We describe only the most common levels here; Figure 11.15 shows them pictorially (in the figure, P indicates error-correcting bits and C indicates a second copy of the data). In all cases depicted in the figure, four drives' worth of data are stored, and the extra drives are used to store redundant information for failure recovery.

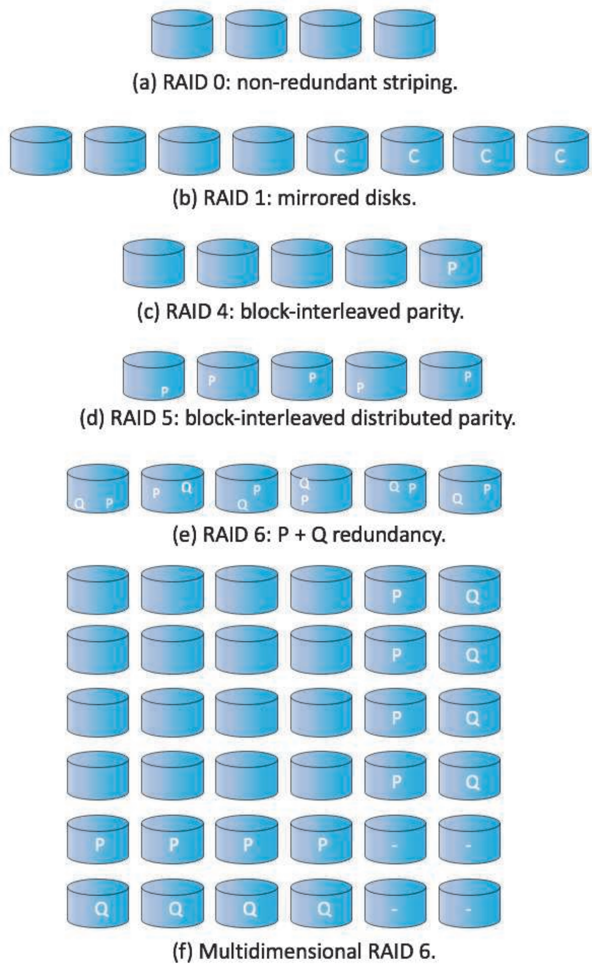


Figure 11.15 RAID levels.

- **RAID level 0**. RAID level 0 refers to drive arrays with striping at the level of blocks but without any redundancy (such as mirroring or parity bits), as shown in Figure 11.15(a).
- **RAID level 1**. RAID level 1 refers to drive mirroring. Figure 11.15(b) shows a mirrored organization.
- **RAID level 4**. RAID level 4 is also known as memory-style error-correcting-code (ECC) organization. ECC is also used in RAID 5 and 6.

  The idea of ECC can be used directly in storage arrays via striping of blocks across drives. For example, the first data block of a sequence of writes can be stored in drive 1, the second block in drive 2, and so on until the Nth block is stored in drive N; the error-correction calculation result of those blocks is stored on drive $N + 1$. This scheme is shown in Figure 11.15(c), where the drive labeled P stores the error-correction block. If one of the drives fails, the error-correction code recalculation detects that and prevents the data from being passed to the requesting process, throwing an error.

RAID 4 can actually correct errors, even though there is only one ECC block. It takes into account the fact that, unlike memory systems, drive controllers can detect whether a sector has been read correctly, so a single parity block can be used for error correction and detection. The idea is as follows: If one of the sectors is damaged, we know exactly which sector it is. We disregard the data in that sector and use the parity data to recalculate the bad data. For every bit in the block, we can determine if it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other drives. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.

A block read accesses only one drive, allowing other requests to be processed by the other drives. The transfer rates for large reads are high, since all the disks can be read in parallel. Large writes also have high transfer rates, since the data and parity can be written in parallel.

Small independent writes cannot be performed in parallel. An operating-system write of data smaller than a block requires that the block be read, modified with the new data, and written back. The parity block has to be updated as well. This is known as the **read-modify-write cycle**. Thus, a single write requires four drive accesses: two to read the two old blocks and two to write the two new blocks.

WAFL (which we cover in Chapter 14) uses RAID level 4 because this RAID level allows drives to be added to a RAID set seamlessly. If the added drives are initialized with blocks containing only zeros, then the parity value does not change, and the RAID set is still correct.

RAID level 4 has two advantages over level 1 while providing equal data protection. First, the storage overhead is reduced because only one parity drive is needed for several regular drives, whereas one mirror drive is needed for every drive in level 1. Second, since reads and writes of a series of blocks are spread out over multiple drives with N-way striping of data, the transfer rate for reading or writing a set of blocks is N times as fast as with level 1.

A performance problem with RAID 4—and with all parity-based RAID levels—is the expense of computing and writing the XOR parity. This overhead can result in slower writes than with non-parity RAID arrays. Modern general-purpose CPUs are very fast compared with drive I/O, however, so the performance hit can be minimal. Also, many RAID storage arrays or host bus-adapters include a hardware controller with dedicated parity hardware. This controller offloads the parity computation from the CPU to the array. The array has an NVRAM cache as well, to store the blocks while the parity is computed and to buffer the writes from the controller to the drives. Such buffering can avoid most read-modify-write cycles by gathering data to be written into a full stripe and writing to all drives in the stripe concurrently. This combination of hardware acceleration and buffering can make parity RAID almost as fast as non-parity RAID, frequently outperforming a non-caching non-parity RAID.

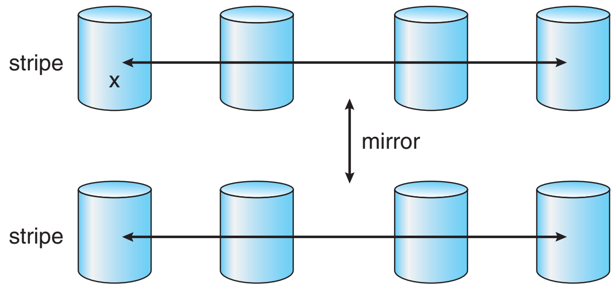
- **RAID level 5**. RAID level 5, or block-interleaved distributed parity, differs from level 4 in that it spreads data and parity among all $N+1$ drives, rather than storing data in N drives and parity in one drive. For each set of N blocks, one of the drives stores the parity and the others store data. For example, with an array of five drives, the parity for the nth block is stored in drive (n mod 5) + 1. The nth blocks of the other four drives store actual data for that block. This setup is shown in Figure 11.15(d), where the Ps are distributed across all the drives. A parity block cannot store parity for blocks in the same drive, because a drive failure would result in loss of data as well as of parity, and hence the loss would not be recoverable. By spreading the parity across all the drives in the set, RAID 5 avoids potential overuse of a single parity drive, which can occur with RAID 4. RAID 5 is the most common parity RAID.

- **RAID level 6**. RAID level 6, also called the **P + Q redundancy scheme**, is much like RAID level 5 but stores extra redundant information to guard against multiple drive failures. XOR parity cannot be used on both parity blocks because they would be identical and would not provide more recovery information. Instead of parity, error-correcting codes such as **Galois fiel math** are used to calculate Q. In the scheme shown in Figure 11.15(e), 2 blocks of redundant data are stored for every 4 blocks of data—compared with 1 parity block in level 5—and the system can tolerate two drive failures.

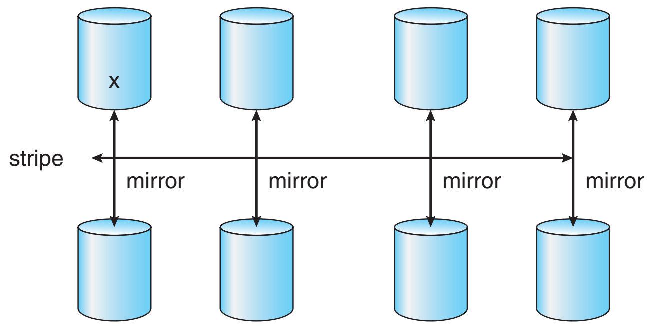
- **Multidimensional RAID level 6**. Some sophisticated storage arrays amplify RAID level 6. Consider an array containing hundreds of drives. Putting those drives in a RAID level 6 stripe would result in many data drives and only two logical parity drives. Multidimensional RAID level 6 logically arranges drives into rows and columns (two or more dimensional arrays) and implements RAID level 6 both horizontally along the rows and vertically down the columns. The system can recover from any failure—or, indeed, multiple failures—by using parity blocks in any of these locations. This RAID level is shown in Figure 11.15(f). For simplicity, the figure shows the RAID parity on dedicated drives, but in reality the RAID blocks are scattered throughout the rows and columns.

- **RAID levels 0 + 1 and 1 + 0**. RAID level 0 + 1 refers to a combination of RAID levels 0 and 1. RAID 0 provides the performance, while RAID 1 provides the reliability. Generally, this level provides better performance than RAID 5. It is common in environments where both performance and reliability are important. Unfortunately, like RAID 1, it doubles the number of drives needed for storage, so it is also relatively expensive. In RAID 0 + 1, a set of drives are striped, and then the stripe is mirrored to another, equivalent stripe.

  Another RAID variation is RAID level 1 + 0, in which drives are mirrored in pairs and then the resulting mirrored pairs are striped. This scheme has some theoretical advantages over RAID 0 + 1. For example, if a single drive fails in RAID 0 + 1, an entire stripe is inaccessible, leaving only the other stripe. With a failure in RAID 1 + 0, a single drive is unavailable, but the drive that mirrors it is still available, as are all the rest of the drives (Figure 11.16).



a) RAID 0 + 1 with a single disk failure.



b) RAID 1 + 0 with a single disk failure.

Figure 11.16 RAID 0 + 1 and 1 + 0 with a single disk failure.

Numerous variations have been proposed to the basic RAID schemes described here. As a result, some confusion may exist about the exact definitions of the different RAID levels.

The implementation of RAID is another area of variation. Consider the following layers at which RAID can be implemented.

- Volume-management software can implement RAID within the kernel or at the system software layer. In this case, the storage hardware can provide minimal features and still be part of a full RAID solution.
- RAID can be implemented in the host bus-adapter (HBA) hardware. Only the drives directly connected to the HBA can be part of a given RAID set. This solution is low in cost but not very flexible.
- RAID can be implemented in the hardware of the storage array. The storage array can create RAID sets of various levels and can even slice these sets into smaller volumes, which are then presented to the operating system. The operating system need only implement the file system on each of the volumes. Arrays can have multiple connections available or can be part of a SAN, allowing multiple hosts to take advantage of the array's features.
- RAID can be implemented in the SAN interconnect layer by drive virtualization devices. In this case, a device sits between the hosts and the storage. It

accepts commands from the servers and manages access to the storage. It could provide mirroring, for example, by writing each block to two separate storage devices.

Other features, such as snapshots and replication, can be implemented at each of these levels as well. A **snapshot** is a view of the file system before the last update took place. (Snapshots are covered more fully in Chapter 14.) **Replication** involves the automatic duplication of writes between separate sites for redundancy and disaster recovery. Replication can be synchronous or asynchronous. In synchronous replication, each block must be written locally and remotely before the write is considered complete, whereas in asynchronous replication, the writes are grouped together and written periodically. Asynchronous replication can result in data loss if the primary site fails, but it is faster and has no distance limitations. Increasingly, replication is also used within a data center or even within a host. As an alternative to RAID protection, replication protects against data loss and also increases read performance (by allowing reads from each of the replica copies). It does of course use more storage than most types of RAID.

The implementation of these features differs depending on the layer at which RAID is implemented. For example, if RAID is implemented in software, then each host may need to carry out and manage its own replication. If replication is implemented in the storage array or in the SAN interconnect, however, then whatever the host operating system or its features, the host's data can be replicated.

One other aspect of most RAID implementations is a hot spare drive or drives. A **hot spare** is not used for data but is configured to be used as a replacement in case of drive failure. For instance, a hot spare can be used to rebuild a mirrored pair should one of the drives in the pair fail. In this way, the RAID level can be reestablished automatically, without waiting for the failed drive to be replaced. Allocating more than one hot spare allows more than one failure to be repaired without human intervention.

11.8.4 Selecting a RAID Level

Given the many choices they have, how do system designers choose a RAID level? One consideration is rebuild performance. If a drive fails, the time needed to rebuild its data can be significant. This may be an important factor if a continuous supply of data is required, as it is in high-performance or interactive database systems. Furthermore, rebuild performance influences the mean time between failures.

Rebuild performance varies with the RAID level used. Rebuilding is easiest for RAID level 1, since data can be copied from another drive. For the other levels, we need to access all the other drives in the array to rebuild data in a failed drive. Rebuild times can be hours for RAID level 5 rebuilds of large drive sets.

RAID level 0 is used in high-performance applications where data loss is not critical. For example, in scientific computing where a data set is loaded and explored, RAID level 0 works well because any drive failures would just require a repair and reloading of the data from its source. RAID level 1 is popular for applications that require high reliability with fast recovery. RAID

THE InServ STORAGE ARRAY

Innovation, in an effort to provide better, faster, and less expensive solutions, frequently blurs the lines that separated previous technologies. Consider the InServ storage array from HP 3Par. Unlike most other storage arrays, InServ does not require that a set of drives be configured at a specific RAID level. Rather, each drive is broken into 256-MB "chunklets." RAID is then applied at the chunklet level. A drive can thus participate in multiple and various RAID levels as its chunklets are used for multiple volumes.

InServ also provides snapshots similar to those created by the WAFL file system. The format of InServ snapshots can be read–write as well as read-only, allowing multiple hosts to mount copies of a given file system without needing their own copies of the entire file system. Any changes a host makes in its own copy are copy-on-write and so are not reflected in the other copies.

A further innovation is **utility storage**. Some file systems do not expand or shrink. On these systems, the original size is the only size, and any change requires copying data. An administrator can configure InServ to provide a host with a large amount of logical storage that initially occupies only a small amount of physical storage. As the host starts using the storage, unused drives are allocated to the host, up to the original logical level. The host thus can believe that it has a large fixed storage space, create its file systems there, and so on. Drives can be added to or removed from the file system by InServ without the file system's noticing the change. This feature can reduce the number of drives needed by hosts, or at least delay the purchase of drives until they are really needed.

0 + 1 and 1 + 0 are used where both performance and reliability are important —for example, for small databases. Due to RAID 1's high space overhead, RAID 5 is often preferred for storing moderate volumes of data. RAID 6 and multidimensional RAID 6 are the most common formats in storage arrays. They offer good performance and protection without large space overhead.

RAID system designers and administrators of storage have to make several other decisions as well. For example, how many drives should be in a given RAID set? How many bits should be protected by each parity bit? If more drives are in an array, data-transfer rates are higher, but the system is more expensive. If more bits are protected by a parity bit, the space overhead due to parity bits is lower, but the chance that a second drive will fail before the first failed drive is repaired is greater, and that will result in data loss.

11.8.5 Extensions

The concepts of RAID have been generalized to other storage devices, including arrays of tapes, and even to the broadcast of data over wireless systems. When applied to arrays of tapes, RAID structures are able to recover data even if one of the tapes in an array is damaged. When applied to broadcast of data, a block of data is split into short units and is broadcast along with a parity unit. If one of the units is not received for any reason, it can be reconstructed from the other units. Commonly, tape-drive robots containing multiple tape drives will stripe data across all the drives to increase throughput and decrease backup time.

11.8.6 Problems with RAID

Unfortunately, RAID does not always assure that data are available for the operating system and its users. A pointer to a file could be wrong, for example, or pointers within the file structure could be wrong. Incomplete writes (called "torn writes"), if not properly recovered, could result in corrupt data. Some other process could accidentally write over a file system's structures, too. RAID protects against physical media errors, but not other hardware and software errors. A failure of the hardware RAID controller, or a bug in the software RAID code, could result in total data loss. As large as is the landscape of software and hardware bugs, that is how numerous are the potential perils for data on a system.

The **Solaris ZFS** file system takes an innovative approach to solving these problems through the use of checksums. ZFS maintains internal checksums of all blocks, including data and metadata. These checksums are not kept with the block that is being checksummed. Rather, they are stored with the pointer to that block. (See Figure 11.17.) Consider an **inode**—a data structure for storing file system metadata—with pointers to its data. Within the inode is the checksum of each block of data. If there is a problem with the data, the checksum will be incorrect, and the file system will know about it. If the data are mirrored, and there is a block with a correct checksum and one with an incorrect checksum, ZFS will automatically update the bad block with the good one. Similarly, the directory entry that points to the inode has a checksum for the inode. Any problem in the inode is detected when the directory is accessed. This checksumming takes places throughout all ZFS structures, providing a much higher level of consistency, error detection, and error cor-

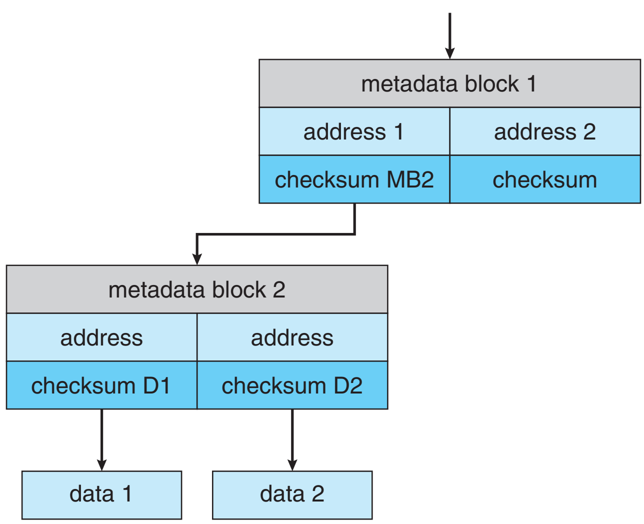


Figure 11.17 ZFS checksums all metadata and data.

rection than is found in RAID drive sets or standard file systems. The extra overhead that is created by the checksum calculation and extra block read-modify-write cycles is not noticeable because the overall performance of ZFS is very fast. (A similar checksum feature is found in the Linux BTRFS file system. See https://btrfs.wiki.kernel.org/index.php/Btrfs_design.)

Another issue with most RAID implementations is lack of flexibility. Consider a storage array with twenty drives divided into four sets of five drives. Each set of five drives is a RAID level 5 set. As a result, there are four separate volumes, each holding a file system. But what if one file system is too large to fit on a five-drive RAID level 5 set? And what if another file system needs very little space? If such factors are known ahead of time, then the drives and volumes can be properly allocated. Very frequently, however, drive use and requirements change over time.

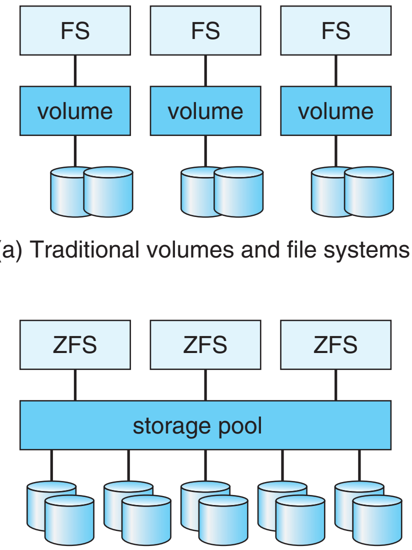
Even if the storage array allowed the entire set of twenty drives to be created as one large RAID set, other issues could arise. Several volumes of various sizes could be built on the set. But some volume managers do not allow us to change a volume's size. In that case, we would be left with the same issue described above—mismatched file-system sizes. Some volume managers allow size changes, but some file systems do not allow for file-system growth or shrinkage. The volumes could change sizes, but the file systems would need to be recreated to take advantage of those changes.

ZFS combines file-system management and volume management into a unit providing greater functionality than the traditional separation of those functions allows. Drives, or partitions of drives, are gathered together via RAID sets into **pools** of storage. A pool can hold one or more ZFS file systems. The entire pool's free space is available to all file systems within that pool. ZFS uses the memory model of `malloc()` and `free()` to allocate and release storage for each file system as blocks are used and freed within the file system. As a result, there are no artificial limits on storage use and no need to relocate file systems between volumes or resize volumes. ZFS provides quotas to limit the size of a file system and reservations to assure that a file system can grow by a specified amount, but those variables can be changed by the file-system owner at any time. Other systems like Linux have volume managers that allow the logical joining of multiple disks to create larger-than-disk volumes to hold large file systems. Figure 11.18(a) depicts traditional volumes and file systems, and Figure 11.18(b) shows the ZFS model.

11.8.7 Object Storage

General-purpose computers typically use file systems to store content for users. Another approach to data storage is to start with a storage pool and place objects in that pool. This approach differs from file systems in that there is no way to navigate the pool and find those objects. Thus, rather than being user-oriented, object storage is computer-oriented, designed to be used by programs. A typical sequence is:

1. Create an object within the storage pool, and receive an object ID.
2. Access the object when needed via the object ID.
3. Delete the object via the object ID.



(a) Traditional volumes and file systems.

(b) ZFS and pooled storage.

Figure 11.18 Traditional volumes and file systems compared with the ZFS model.

Object storage management software, such as the **Hadoop fil system** (**HDFS**) and **Ceph**, determines where to store the objects and manages object protection. Typically, this occurs on commodity hardware rather than RAID arrays. For example, HDFS can store *N* copies of an object on *N* different computers. This approach can be lower in cost than storage arrays and can provide fast access to that object (at least on those *N* systems). All systems in a Hadoop cluster can access the object, but only systems that have a copy have fast access via the copy. Computations on the data occur on those systems, with results sent across the network, for example, only to the systems requesting them. Other systems need network connectivity to read and write to the object. Therefore, object storage is usually used for bulk storage, not high-speed random access. Object storage has the advantage of **horizontal scalability**. That is, whereas a storage array has a fixed maximum capacity, to add capacity to an object store, we simply add more computers with internal disks or attached external disks and add them to the pool. Object storage pools can be petabytes in size.

Another key feature of object storage is that each object is self-describing, including description of its contents. In fact, object storage is also known as **content-addressable storage**, because objects can be retrieved based on their contents. There is no set format for the contents, so what the system stores is **unstructured data**.

While object storage is not common on general-purpose computers, huge amounts of data are stored in object stores, including Google's Internet search contents, Dropbox contents, Spotify's songs, and Facebook photos. Cloud computing (such as Amazon AWS) generally uses object stores (in Amazon S3) to hold file systems as well as data objects for customer applications running on cloud computers.

For the history of object stores see http://www.theregister.co.uk/2016/07/15/the_history_boys_cas_and_object_storage_map.

11.9 Summary

- Hard disk drives and nonvolatile memory devices are the major secondary storage I/O units on most computers. Modern secondary storage is structured as large one-dimensional arrays of logical blocks.
- Drives of either type may be attached to a computer system in one of three ways: (1) through the local I/O ports on the host computer, (2) directly connected to motherboards, or (3) through a communications network or storage network connection.
- Requests for secondary storage I/O are generated by the file system and by the virtual memory system. Each request specifies the address on the device to be referenced in the form of a logical block number.
- Disk-scheduling algorithms can improve the effective bandwidth of HDDs, the average response time, and the variance in response time. Algorithms such as SCAN and C-SCAN are designed to make such improvements through strategies for disk-queue ordering. Performance of disk-scheduling algorithms can vary greatly on hard disks. In contrast, because solid-state disks have no moving parts, performance varies little among scheduling algorithms, and quite often a simple FCFS strategy is used.
- Data storage and transmission are complex and frequently result in errors. Error detection attempts to spot such problems to alert the system for corrective action and to avoid error propagation. Error correction can detect and repair problems, depending on the amount of correction data available and the amount of data that was corrupted.
- Storage devices are partitioned into one or more chunks of space. Each partition can hold a volume or be part of a multidevice volume. File systems are created in volumes.
- The operating system manages the storage device's blocks. New devices typically come pre-formatted. The device is partitioned, file systems are created, and boot blocks are allocated to store the system's bootstrap program if the device will contain an operating system. Finally, when a block or page is corrupted, the system must have a way to lock out that block or to replace it logically with a spare.
- An efficient swap space is a key to good performance in some systems. Some systems dedicate a raw partition to swap space, and others use a file within the file system instead. Still other systems allow the user or system administrator to make the decision by providing both options.
- Because of the amount of storage required on large systems, and because storage devices fail in various ways, secondary storage devices are frequently made redundant via RAID algorithms. These algorithms allow more than one drive to be used for a given operation and allow continued

operation and even automatic recovery in the face of a drive failure. RAID algorithms are organized into different levels; each level provides some combination of reliability and high transfer rates.

- Object storage is used for big data problems such as indexing the Internet and cloud photo storage. Objects are self-defining collections of data, addressed by object ID rather than file name. Typically it uses replication for data protection, computes based on the data on systems where a copy of the data exists, and is horizontally scalable for vast capacity and easy expansion.

Practice Exercises

11.1 Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer.

11.2 Explain why SSTF scheduling tends to favor middle cylinders over the innermost and outermost cylinders.

11.3 Why is rotational latency usually not considered in disk scheduling? How would you modify SSTF, SCAN, and C-SCAN to include latency optimization?

11.4 Why is it important to balance file-system I/O among the disks and controllers on a system in a multitasking environment?

11.5 What are the tradeoffs involved in rereading code pages from the file system versus using swap space to store them?

11.6 Is there any way to implement truly stable storage? Explain your answer.

11.7 It is sometimes said that tape is a sequential-access medium, whereas a hard disk is a random-access medium. In fact, the suitability of a storage device for random access depends on the transfer size. The term ***streaming transfer rate*** denotes the rate for a data transfer that is underway, excluding the effect of access latency. In contrast, the ***effective transfer rate*** is the ratio of total bytes to total seconds, including overhead time such as access latency.

Suppose we have a computer with the following characteristics: the level-2 cache has an access latency of 8 nanoseconds and a streaming transfer rate of 800 megabytes per second, the main memory has an access latency of 60 nanoseconds and a streaming transfer rate of 80 megabytes per second, the hard disk has an access latency of 15 milliseconds and a streaming transfer rate of 5 megabytes per second, and a tape drive has an access latency of 60 seconds and a streaming transfer rate of 2 megabytes per second.

a. Random access causes the effective transfer rate of a device to decrease, because no data are transferred during the access time. For the disk described, what is the effective transfer rate if an average access is followed by a streaming transfer of (1) 512 bytes, (2) 8 kilobytes, (3) 1 megabyte, and (4) 16 megabytes?

b. The utilization of a device is the ratio of effective transfer rate to streaming transfer rate. Calculate the utilization of the disk drive for each of the four transfer sizes given in part a.

c. Suppose that a utilization of 25 percent (or higher) is considered acceptable. Using the performance figures given, compute the smallest transfer size for a disk that gives acceptable utilization.

d. Complete the following sentence: A disk is a random-access device for transfers larger than ______ bytes and is a sequential-access device for smaller transfers.

e. Compute the minimum transfer sizes that give acceptable utilization for cache, memory, and tape.

f. When is a tape a random-access device, and when is it a sequential-access device?

11.8 Could a RAID level 1 organization achieve better performance for read requests than a RAID level 0 organization (with nonredundant striping of data)? If so, how?

11.9 Give three reasons to use HDDs as secondary storage.

11.10 Give three reasons to use NVM devices as secondary storage.

Further Reading

[Services (2012)] provides an overview of data storage in a variety of modern computing environments. Discussions of redundant arrays of independent disks (RAIDs) are presented by [Patterson et al. (1988)]. [Kim et al. (2009)] discuss disk-scheduling algorithms for SSDs. Object-based storage is described by [Mesnier et al. (2003)].

[Russinovich et al. (2017)], [McDougall and Mauro (2007)], and [Love (2010)] discuss file-system details in Windows, Solaris, and Linux, respectively.

Storage devices are continuously evolving, with goals of increasing performance, increasing capacity, or both. For one direction in capacity improvement see http://www.tomsitpro.com/articles/shingled-magnetic-recoding-smr-101-basics,2-933.html).

RedHat (and other) Linux distributions have multiple, selectable disk scheduling algorithms. For details see https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Performance_Tuning_Guide/index.html.

Learn more about the default Linux bootstrap loader at https://www.gnu.org/software/grub/manual/grub.html/.

A relatively new file system, BTRFS, is detailed in https://btrfs.wiki.kernel.org/index.php/Btrfs_design.

For the history of object stores see http://www.theregister.co.uk/2016/07/15/the_history_boys_cas_and_object_storage_map.

Bibliography

[Kim et al. (2009)] J. Kim, Y. Oh, E. Kim, J. C. D. Lee, and S. Noh, "Disk Schedulers for Solid State Drivers", *Proceedings of the seventh ACM international conference on Embedded software* (2009), pages 295–304.

[Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

[McDougall and Mauro (2007)] R. McDougall and J. Mauro, *Solaris Internals*, Second Edition, Prentice Hall (2007).

[Mesnier et al. (2003)] M. Mesnier, G. Ganger, and E. Ridel, "Object-based storage", *IEEE Communications Magazine*, Volume 41, Number 8 (2003), pages 84–99.

[Patterson et al. (1988)] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (1988), pages 109–116.

[Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1*, Seventh Edition, Microsoft Press (2017).

[Services (2012)] E. E. Services, *Information Storage and Management: Storing, Managing, and Protecting Digital Information in Classic, Virtualized, and Cloud Environments*, Wiley (2012).

Chapter 11 Exercises

11.11 None of the disk-scheduling disciplines, except FCFS, is truly fair (starvation may occur).

a. Explain why this assertion is true.

b. Describe a way to modify algorithms such as SCAN to ensure fairness.

c. Explain why fairness is an important goal in a multi-user systems.

d. Give three or more examples of circumstances in which it is important that the operating system be unfair in serving I/O requests.

11.12 Explain why NVM devices often use an FCFS disk-scheduling algorithm.

11.13 Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4,999. The drive is currently serving a request at cylinder 2,150, and the previous request was at cylinder 1,805. The queue of pending requests, in FIFO order, is:

2,069; 1,212; 2,296; 2,800; 544; 1,618; 356; 1,523; 4,965; 3,681

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

a. FCFS

b. SCAN

c. C-SCAN

11.14 Elementary physics states that when an object is subjected to a constant acceleration a, the relationship between distance d and time t is given by $d = \frac{1}{2}at^2$. Suppose that, during a seek, the disk in Exercise 11.14 accelerates the disk arm at a constant rate for the first half of the seek, then decelerates the disk arm at the same rate for the second half of the seek. Assume that the disk can perform a seek to an adjacent cylinder in 1 millisecond and a full-stroke seek over all 5,000 cylinders in 18 milliseconds.

a. The distance of a seek is the number of cylinders over which the head moves. Explain why the seek time is proportional to the square root of the seek distance.

b. Write an equation for the seek time as a function of the seek distance. This equation should be of the form $t = x + y\sqrt{L}$, where t is the time in milliseconds and L is the seek distance in cylinders.

c. Calculate the total seek time for each of the schedules in Exercise 11.14. Determine which schedule is the fastest (has the smallest total seek time).

d. The **percentage speedup** is the time saved divided by the original time. What is the percentage speedup of the fastest schedule over FCFS?

11.15 Suppose that the disk in Exercise 11.15 rotates at 7,200 RPM.

a. What is the average rotational latency of this disk drive?

b. What seek distance can be covered in the time that you found for part a?

11.16 Compare and contrast HDDs and NVM devices. What are the best applications for each type?

11.17 Describe some advantages and disadvantages of using NVM devices as a caching tier and as a disk-drive replacement compared with using only HDDs.

11.18 Compare the performance of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests. Consider the average response time (the time between the arrival of a request and the completion of that request's service), the variation in response time, and the effective bandwidth. How does performance depend on the relative sizes of seek time and rotational latency?

11.19 Requests are not usually uniformly distributed. For example, we can expect a cylinder containing the file-system metadata to be accessed more frequently than a cylinder containing only files. Suppose you know that 50 percent of the requests are for a small, fixed number of cylinders.

a. Would any of the scheduling algorithms discussed in this chapter be particularly good for this case? Explain your answer.

b. Propose a disk-scheduling algorithm that gives even better performance by taking advantage of this "hot spot" on the disk.

11.20 Consider a RAID level 5 organization comprising five disks, with the parity for sets of four blocks on four disks stored on the fifth disk. How many blocks are accessed in order to perform the following?

a. A write of one block of data

b. A write of seven continuous blocks of data

11.21 Compare the throughput achieved by a RAID level 5 organization with that achieved by a RAID level 1 organization for the following:

a. Read operations on single blocks

b. Read operations on multiple contiguous blocks

11.22 Compare the performance of write operations achieved by a RAID level 5 organization with that achieved by a RAID level 1 organization.

11.23 Assume that you have a mixed configuration comprising disks organized as RAID level 1 and RAID level 5 disks. Assume that the system has flexibility in deciding which disk organization to use for storing a particular file. Which files should be stored in the RAID level 1 disks and which in the RAID level 5 disks in order to optimize performance?

11.24 The reliability of a storage device is typically described in terms of mean time between failures (MTBF). Although this quantity is called a "time," the MTBF actually is measured in drive-hours per failure.

a. If a system contains 1,000 disk drives, each of which has a 750,000-hour MTBF, which of the following best describes how often a drive failure will occur in that disk farm: once per thousand years, once per century, once per decade, once per year, once per month, once per week, once per day, once per hour, once per minute, or once per second?

b. Mortality statistics indicate that, on the average, a U.S. resident has about 1 chance in 1,000 of dying between the ages of 20 and 21. Deduce the MTBF hours for 20-year-olds. Convert this figure from hours to years. What does this MTBF tell you about the expected lifetime of a 20-year-old?

c. The manufacturer guarantees a 1-million-hour MTBF for a certain model of disk drive. What can you conclude about the number of years for which one of these drives is under warranty?

11.25 Discuss the relative advantages and disadvantages of sector sparing and sector slipping.

11.26 Discuss the reasons why the operating system might require accurate information on how blocks are stored on a disk. How could the operating system improve file-system performance with this knowledge?

Programming Problems

11.27 Write a program that implements the following disk-scheduling algorithms:

a. FCFS

b. SCAN

c. C-SCAN

Your program will service a disk with 5,000 cylinders numbered 0 to 4,999. The program will generate a random series of 1,000 cylinder requests and service them according to each of the algorithms listed above. The program will be passed the initial position of the disk head (as a parameter on the command line) and report the total amount of head movement required by each algorithm.

CHAPTER 12

I/O Systems

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

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

CHAPTER OBJECTIVES

- Explore the structure of an operating system's I/O subsystem.
- Discuss the principles and complexities of I/O hardware.
- Explain the performance aspects of I/O hardware and software.

12.1 Overview

The control of devices connected to the computer is a major concern of operating-system designers. Because I/O devices vary so widely in their function and speed (consider a mouse, a hard disk, a flash drive, and a tape robot), varied methods are needed to control them. These methods form the I/O subsystem of the kernel, which separates the rest of the kernel from the complexities of managing I/O devices.

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

12.2 I/O Hardware

Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks, tapes), transmission devices (network connections, Bluetooth), and human-interface devices (screen, keyboard, mouse, audio in and out). Other devices are more specialized, such as those involved in the steering of a jet. In these aircraft, a human gives input to the flight computer via a joystick and foot pedals, and the computer sends output commands that cause motors to move rudders and flaps and fuels to the engines. Despite the incredible variety of I/O devices, though, we need only a few concepts to understand how the devices are attached and how the software can control the hardware.

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point, or **port**—for example, a serial port. (The term **PHY**, shorthand for the OSI model physical layer, is also used in reference to ports but is more common in data-center nomenclature.) If devices share a common set of wires, the connection is called a bus. A **bus**, like the PCI bus used in most computers today, is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings. When device *A* has a cable that plugs into device *B*, and device *B* has a cable that plugs into device *C*, and device *C* plugs into a port on the computer, this arrangement is called a **daisy chain**. A daisy chain usually operates as a bus.

Buses are used widely in computer architecture and vary in their signaling methods, speed, throughput, and connection methods. A typical PC bus structure appears in Figure 12.1. In the figure, a **PCIe bus** (the common PC system bus) connects the processor–memory subsystem to fast devices, and an **expansion bus** connects relatively slow devices, such as the keyboard and serial and USB ports. In the lower-left portion of the figure, four disks are connected together on a **serial-attached SCSI** (**SAS**) bus plugged into an SAS controller. PCIe is a flexible bus that sends data over one or more "lanes." A lane is composed of two signaling pairs, one pair for receiving data and the other for transmitting. Each lane is therefore composed of four wires, and each

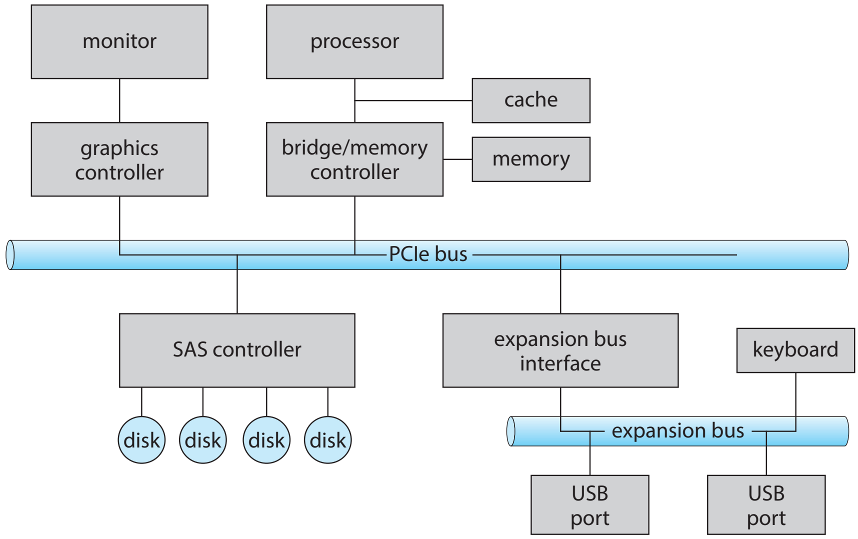


Figure 12.1 A typical PC bus structure.

lane is used as a full-duplex byte stream, transporting data packets in an eight-bit byte format simultaneously in both directions. Physically, PCIe links may contain 1, 2, 4, 8, 12, 16, or 32 lanes, as signified by an "x" prefix. A PCIe card or connector that uses 8 lanes is designated x8, for example. In addition, PCIe has gone through multiple "generations," with more coming in the future. Thus, for example, a card might be "PCIe gen3 x8", which means it works with generation 3 of PCIe and uses 8 lanes. Such a device has maximum throughput of 8 gigabytes per second. Details about PCIe can be found at https://pcisig.com.

A **controller** is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port. By contrast, a **fibr channel** (**FC**) bus controller is not simple. Because the FC protocol is complex and used in data centers rather than on PCs, the FC bus controller is often implemented as a separate circuit board —or a **host bus adapter** (**HBA**)—that connects to a bus in the computer. It typically contains a processor, microcode, and some private memory to enable it to process the FC protocol messages. Some devices have their own built-in controllers. If you look at a disk drive, you will see a circuit board attached to one side. This board is the disk controller. It implements the disk side of the protocol for some kinds of connection—SAS and SATA, for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, and caching.

12.2.1 Memory-Mapped I/O

How does the processor give commands and data to a controller to accomplish an I/O transfer? The short answer is that the controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way in which this communication can occur is through the use of special I/O instructions

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Figure 12.2 Device I/O port locations on PCs (partial).

that specify the transfer of a byte or a word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, the device can support **memory-mapped I/O**. In this case, the device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers at their mapped locations in physical memory.

In the past, PCs often used I/O instructions to control some devices and memory-mapped I/O to control others. Figure 12.2 shows the usual I/O port addresses for PCs. The graphics controller has I/O ports for basic control operations, but the controller has a large memory-mapped region to hold screen contents. A thread sends output to the screen by writing data into the memory-mapped region. The controller generates the screen image based on the contents of this memory. This technique is simple to use. Moreover, writing millions of bytes to the graphics memory is faster than issuing millions of I/O instructions. Therefore, over time, systems have moved toward memory-mapped I/O. Today, most I/O is performed by device controllers using memory-mapped I/O.

I/O device control typically consists of four registers, called the status, control, data-in, and data-out registers.

- The **data-in register** is read by the host to get input.
- The **data-out register** is written by the host to send output.
- The **status register** contains bits that can be read by the host. These bits indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
- The **control register** can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex com-

munication, another bit enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.

The data registers are typically 1 to 4 bytes in size. Some controllers have FIFO chips that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register. A FIFO chip can hold a small burst of data until the device or host is able to receive those data.

12.2.2 Polling

The complete protocol for interaction between the host and a controller can be intricate, but the basic handshaking notion is simple. We explain handshaking with an example. Assume that 2 bits are used to coordinate the producer–consumer relationship between the controller and the host. The controller indicates its state through the `busy` bit in the `status` register. (Recall that to ***set*** a bit means to write a 1 into the bit and to ***clear*** a bit means to write a 0 into it.) The controller sets the `busy` bit when it is busy working and clears the `busy` bit when it is ready to accept the next command. The host signals its wishes via the `command-ready` bit in the `command` register. The host sets the `command-ready` bit when a command is available for the controller to execute. For this example, the host writes output through a port, coordinating with the controller by handshaking as follows.

1. The host repeatedly reads the `busy` bit until that bit becomes clear.
2. The host sets the `write` bit in the `command` register and writes a byte into the `data-out` register.
3. The host sets the `command-ready` bit.
4. When the controller notices that the `command-ready` bit is set, it sets the `busy` bit.
5. The controller reads the command register and sees the `write` command. It reads the `data-out` register to get the byte and does the I/O to the device.
6. The controller clears the `command-ready` bit, clears the `error` bit in the status register to indicate that the device I/O succeeded, and clears the `busy` bit to indicate that it is finished.

This loop is repeated for each byte.

In step 1, the host is **busy-waiting** or **polling**: it is in a loop, reading the `status` register over and over until the `busy` bit becomes clear. If the controller and device are fast, this method is a reasonable one. But if the wait may be long, the host should probably switch to another task. How, then, does the host know when the controller has become idle? For some devices, the host must service the device quickly, or data will be lost. For instance, when data are streaming in on a serial port or from a keyboard, the small buffer on the controller will overflow and data will be lost if the host waits too long before returning to read the bytes.

In many computer architectures, three CPU-instruction cycles are sufficient to poll a device: `read` a device register, `logical-and` to extract a status bit, and `branch` if not zero. Clearly, the basic polling operation is efficient. But polling becomes inefficient when it is attempted repeatedly yet rarely finds a device ready for service, while other useful CPU processing remains undone. In such instances, it may be more efficient to arrange for the hardware controller to notify the CPU when the device becomes ready for service, rather than to require the CPU to poll repeatedly for an I/O completion. The hardware mechanism that enables a device to notify the CPU is called an **interrupt**.

12.2.3 Interrupts

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-request line, the CPU performs a state save and jumps to the **interrupt-handler routine** at a fixed address in memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a `return from interrupt` instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller ***raises*** an interrupt by asserting a signal on the interrupt request line, the CPU ***catches*** the interrupt and ***dispatches*** it to the interrupt

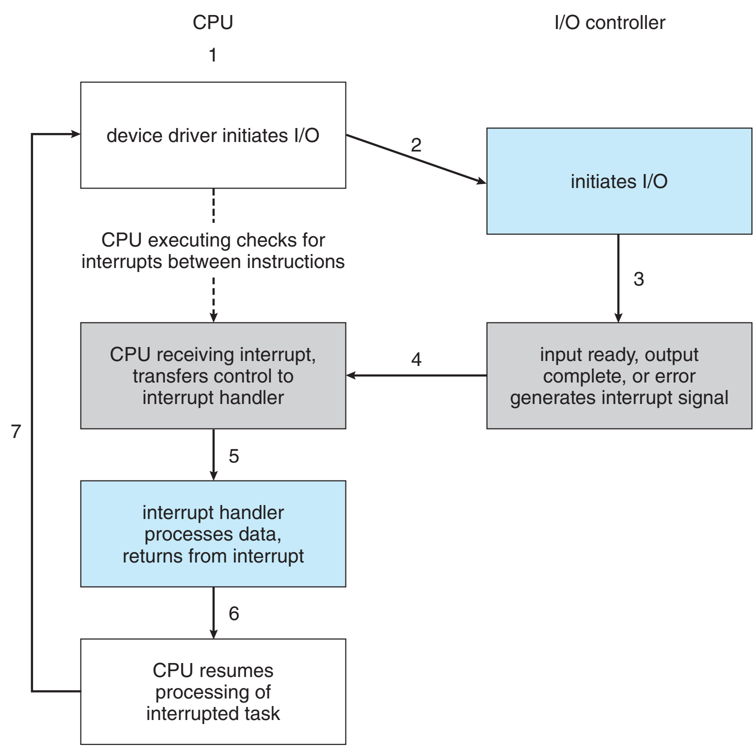


Figure 12.3 Interrupt-driven I/O cycle.

```
Fri Nov 25 13:55:59                                  0:00:10
                         SCHEDULER      INTERRUPTS
------------------------------------------------------
total_samples                   13           22998

delays <  10 usecs              12           16243
delays <  20 usecs               1            5312
delays <  30 usecs               0             473
delays <  40 usecs               0             590
delays <  50 usecs               0              61
delays <  60 usecs               0             317
delays <  70 usecs               0               2
delays <  80 usecs               0               0
delays <  90 usecs               0               0
delays < 100 usecs               0               0
total  < 100 usecs              13           22998
```

Figure 12.4 Latency command on Mac OS X.

handler, and the handler ***clears*** the interrupt by servicing the device. Figure 12.3 summarizes the interrupt-driven I/O cycle.

We stress interrupt management in this chapter because even single-user modern systems manage hundreds of interrupts per second and servers hundreds of thousands per second. For example, Figure 12.4 shows the `latency` command output on macOS, revealing that over ten seconds a quiet desktop computer performed almost 23,000 interrupts.

The basic interrupt mechanism just described enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. In a modern operating system, however, we need more sophisticated interrupt-handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency when there are multiple concurrent interrupts.
4. We need a way for an instruction to get the operating system's attention directly (separately from I/O requests), for activities such as page faults and errors such as division by zero. As we shall see, this task is accomplished by "traps."

In modern computer hardware, these features are provided by the CPU and by the **interrupt-controller hardware**.

Most CPUs have two interrupt request lines. One is the **nonmaskable interrupt**, which is reserved for events such as unrecoverable memory errors. The second interrupt line is **maskable**: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

The interrupt mechanism accepts an **address**—a number that selects a specific interrupt-handling routine from a small set. In most architectures, this address is an offset in a table called the **interrupt vector**. This vector contains the memory addresses of specialized interrupt handlers. The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector. A common way to solve this problem is to use **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Figure 12.5 illustrates the design of the interrupt vector for the Intel Pentium processor. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions (which cause system crashes), page faults (needing immediate action), and debugging requests (stopping normal operation and jumping to a debugger application). The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 12.5 Intel Pentium processor event-vector table.

The interrupt mechanism also implements a system of **interrupt priority levels**. These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and make it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.

A modern operating system interacts with the interrupt mechanism in several ways. At boot time, the operating system probes the hardware buses to determine what devices are present and installs the corresponding interrupt handlers into the interrupt vector. During I/O, the various device controllers raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected. The interrupt mechanism is also used to handle a wide variety of **exceptions**, such as dividing by zero, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode. The events that trigger interrupts have a common property: they are occurrences that induce the operating system to execute an urgent, self-contained routine.

Because interrupt handing in many cases is time and resource constrained and therefore complicated to implement, systems frequently split interrupt management between a **first-leve interrupt handler** (**FLIH**) and a **second-level interrupt handler** (**SLIH**). The FLIH performs the context switch, state storage, and queuing of a handling operation, while the separately scheduled SLIH performs the handling of the requested operation.

Operating systems have other good uses for interrupts as well. For example, many operating systems use the interrupt mechanism for virtual memory paging. A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel. This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt.

Another example is found in the implementation of system calls. Usually, a program uses library calls to issue system calls. The library routines check the arguments given by the application, build a data structure to convey the arguments to the kernel, and then execute a special instruction called a **software interrupt**, or **trap**. This instruction has an operand that identifies the desired kernel service. When a process executes the trap instruction, the interrupt hardware saves the state of the user code, switches to kernel mode, and dispatches to the kernel routine or thread that implements the requested service. The trap is given a relatively low interrupt priority compared with those assigned to device interrupts—executing a system call on behalf of an application is less urgent than servicing a device controller before its FIFO queue overflows and loses data.

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

A threaded kernel architecture is well suited to implement multiple interrupt priorities and to enforce the precedence of interrupt handling over background processing in kernel and application routines. We illustrate this point with the Solaris kernel. In Solaris, interrupt handlers are executed as kernel threads. A range of high scheduling priorities is reserved for these threads. These priorities give interrupt handlers precedence over application code and kernel housekeeping and implement the priority relationships among interrupt handlers. The priorities cause the Solaris thread scheduler to preempt low-priority interrupt handlers in favor of higher-priority ones, and the threaded implementation enables multiprocessor hardware to run several interrupt handlers concurrently. We describe the interrupt architecture of Linux in Chapter 20, Windows10 in Chapter 21, and UNIX in Appendix C.

In summary, interrupts are used throughout modern operating systems to handle asynchronous events and to trap to supervisor-mode routines in the kernel. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Device controllers, hardware faults, and system calls all raise interrupts to trigger kernel routines. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system performance. Interrupt-driven I/O is now much more common than polling, with polling being used for high-throughput I/O. Sometimes the two are used together. Some device drivers use interrupts when the I/O rate is low and switch to polling when the rate increases to the point where polling is faster and more efficient.

12.2.4 Direct Memory Access

For a device that does large transfers, such as a disk drive, it seems wasteful to use an expensive general-purpose processor to watch status bits and to feed data into a controller register one byte at a time—a process termed **programmed I/O** (**PIO**). Computers avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a **direct-memory-access** (**DMA**) controller. To initiate a DMA transfer, the host writes a DMA command block into memory. This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred. A command block can be more complex, including a list of sources and destinations addresses that are not contiguous. This **scatter–gather** method allows multiple transfers to be executed via a single DMA command. The CPU writes the address of this command block to the DMA controller, then goes on with other work. The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU. A simple DMA controller is a standard component in all modern computers, from smartphones to mainframes.

Note that it is most straightforward for the target address to be in kernel address space. If it were in user space, the user could, for example, modify the contents of that space during the transfer, losing some set of data. To get the DMA-transferred data to the user space for thread access, however, a second copy operation, this time from kernel memory to user memory, is needed. This **double buffering** is inefficient. Over time, operating systems have moved to using memory-mapping (see Section 12.2.1) to perform I/O transfers directly between devices and user address space.

Handshaking between the DMA controller and the device controller is performed via a pair of wires called **DMA-request** and **DMA-acknowledge**. The device controller places a signal on the DMA-request wire when a word of data is available for transfer. This signal causes the DMA controller to seize the memory bus, place the desired address on the memory-address wire, and place a signal on the DMA-acknowledge wire. When the device controller receives the DMA-acknowledge signal, it transfers the word of data to memory and removes the DMA-request signal.

When the entire transfer is finished, the DMA controller interrupts the CPU. This process is depicted in Figure 12.6. When the DMA controller seizes the memory bus, the CPU is momentarily prevented from accessing main memory, although it can still access data items in its caches. Although this **cycle stealing** can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the total system performance. Some computer architectures use physical memory addresses for DMA, but

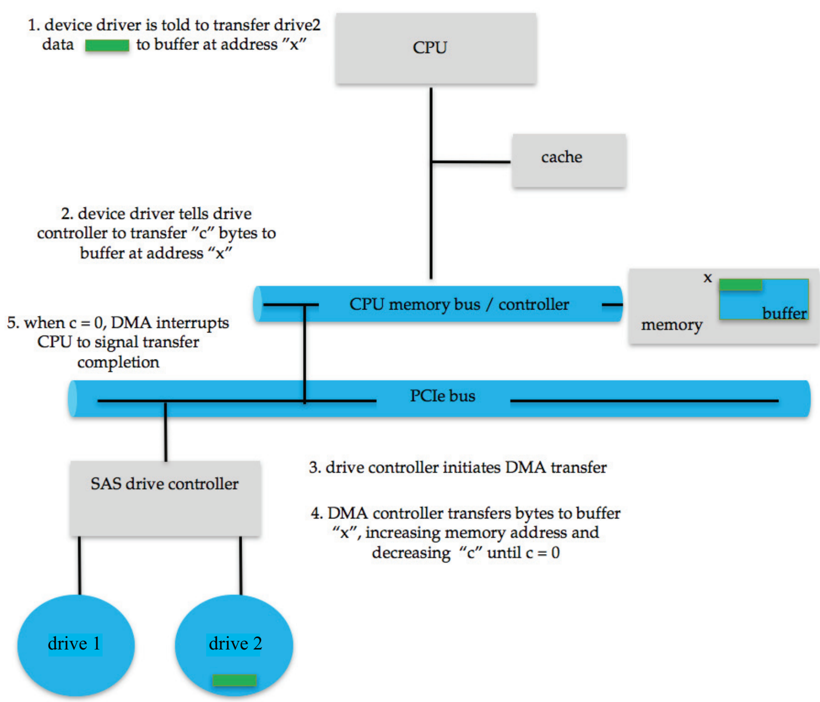


Figure 12.6 Steps in a DMA transfer.

others perform **direct virtual memory access** (**DVMA**), using virtual addresses that undergo translation to physical addresses. DVMA can perform a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory.

On protected-mode kernels, the operating system generally prevents processes from issuing device commands directly. This discipline protects data from access-control violations and also protects the system from erroneous use of device controllers, which could cause a system crash. Instead, the operating system exports functions that a sufficiently privileged process can use to access low-level operations on the underlying hardware. On kernels without memory protection, processes can access device controllers directly. This direct access can be used to achieve high performance, since it can avoid kernel communication, context switches, and layers of kernel software. Unfortunately, it interferes with system security and stability. Common general-purpose operating systems protect memory and devices so that the system can try to guard against erroneous or malicious applications.

12.2.5 I/O Hardware Summary

Although the hardware aspects of I/O are complex when considered at the level of detail of electronics-hardware design, the concepts that we have just described are sufficient to enable us to understand many I/O features of operating systems. Let's review the main concepts:

- A bus
- A controller
- An I/O port and its registers
- The handshaking relationship between the host and a device controller
- The execution of this handshaking in a polling loop or via interrupts
- The offloading of this work to a DMA controller for large transfers

We gave a basic example of the handshaking that takes place between a device controller and the host earlier in this section. In reality, the wide variety of available devices poses a problem for operating-system implementers. Each kind of device has its own set of capabilities, control-bit definitions, and protocols for interacting with the host—and they are all different. How can the operating system be designed so that we can attach new devices to the computer without rewriting the operating system? And when the devices vary so widely, how can the operating system give a convenient, uniform I/O interface to applications? We address those questions next.

12.3 Application I/O Interface

In this section, we discuss structuring techniques and interfaces for the operating system that enable I/O devices to be treated in a standard, uniform way. We explain, for instance, how an application can open a file on a disk without

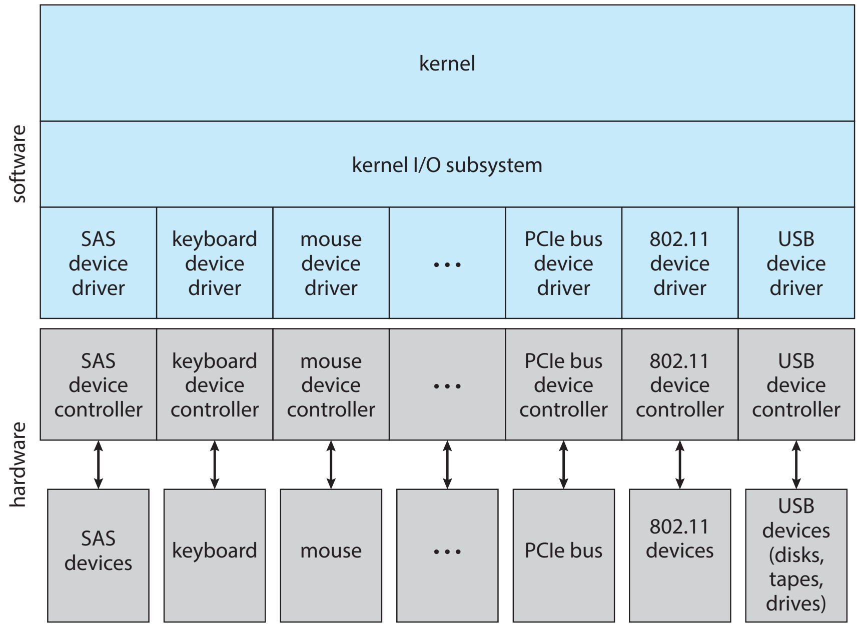


Figure 12.7 A kernel I/O structure.

knowing what kind of disk it is and how new disks and other devices can be added to a computer without disruption of the operating system.

Like other complex software-engineering problems, the approach here involves abstraction, encapsulation, and software layering. Specifically, we can abstract away the detailed differences in I/O devices by identifying a few general kinds. Each general kind is accessed through a standardized set of functions—an **interface**. The differences are encapsulated in kernel modules called device drivers that internally are custom-tailored to specific devices but that export one of the standard interfaces. Figure 12.7 illustrates how the I/O-related portions of the kernel are structured in software layers.

The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel, much as the I/O system calls encapsulate the behavior of devices in a few generic classes that hide hardware differences from applications. Making the I/O subsystem independent of the hardware simplifies the job of the operating-system developer. It also benefits the hardware manufacturers. They either design new devices to be compatible with an existing host controller interface (such as SATA), or they write device drivers to interface the new hardware to popular operating systems. Thus, we can attach new peripherals to a computer without waiting for the operating-system vendor to develop support code.

Unfortunately for device-hardware manufacturers, each type of operating system has its own standards for the device-driver interface. A given device may ship with multiple device drivers—for instance, drivers for Windows, Linux, AIX, and macOS. Devices vary on many dimensions, as illustrated in Figure 12.8.

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Figure 12.8 Characteristics of I/O devices.

- **Character-stream or block**. A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.
- **Sequential or random access**. A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.
- **Synchronous or asynchronous**. A synchronous device performs data transfers with predictable response times, in coordination with other aspects of the system. An asynchronous device exhibits irregular or unpredictable response times not coordinated with other computer events.
- **Sharable or dedicated**. A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.
- **Speed of operation**. Device speeds range from a few bytes per second to gigabytes per second.
- **Read–write, read only, write once**. Some devices perform both input and output, but others support only one data transfer direction. Some allow data to be modified after write, but others can be written only once and are read-only thereafter.

For the purpose of application access, many of these differences are hidden by the operating system, and the devices are grouped into a few conventional types. The resulting styles of device access have been found to be useful and broadly applicable. Although the exact system calls may differ across operating systems, the device categories are fairly standard. The major access conventions include block I/O, character-stream I/O, memory-mapped file access, and network sockets. Operating systems also provide special system calls to access a few additional devices, such as a time-of-day clock and a timer. Some operating systems provide a set of system calls for graphical display, video, and audio devices.

Most operating systems also have an **escape** (or **back door**) that transparently passes arbitrary commands from an application to a device driver. In UNIX, this system call is `ioctl()` (for "I/O control"). The `ioctl()` system call enables an application to access any functionality that can be implemented by any device driver, without the need to invent a new system call. The `ioctl()` system call has three arguments. The first is a device identifier that connects the application to the driver by referring to a hardware device managed by that driver. The second is an integer that selects one of the commands implemented in the driver. The third is a pointer to an arbitrary data structure in memory that enables the application and driver to communicate any necessary control information or data.

The device identifier in UNIX and Linux is a tuple of "major and minor" device numbers. The major number is the device type, and the second is the instance of that device. For example, consider these SSD devices on a system. If one issues a command:

```
% ls -l /dev/sda*
```

then the following output

```
brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda
brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1
brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2
brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3
```

shows that 8 is the major device number. The operating system uses that information to route I/O requests to the appropriate device driver. The minor numbers 0, 1, 2, and 3 indicate the instance of the device, allowing requests for I/O to a device entry to select the exact device for the request.

12.3.1 Block and Character Devices

The **block-device interface** captures all the aspects necessary for accessing disk drives and other block-oriented devices. The device is expected to understand commands such as `read()` and `write()`. If it is a random-access device, it is also expected to have a `seek()` command to specify which block to transfer next. Applications normally access such a device through a file-system interface. We can see that `read()`, `write()`, and `seek()` capture the essential behaviors of block-storage devices, so that applications are insulated from the low-level differences among those devices.

The operating system itself, as well as special applications such as database-management systems, may prefer to access a block device as a simple linear array of blocks. This mode of access is sometimes called **raw I/O**. If the application performs its own buffering, then using a file system would cause extra, unneeded buffering. Likewise, if an application provides its own locking of blocks or regions, then any operating-system locking services would be redundant at the least and contradictory at the worst. To avoid these conflicts, raw-device access passes control of the device directly to the application, letting the operating system step out of the way. Unfortunately, no operating-system services are then performed on this device. A compromise that is becoming common is for the operating system to allow a mode of operation on a file that disables buffering and locking. In the UNIX world, this is called **direct I/O**.

Memory-mapped file access can be layered on top of block-device drivers. Rather than offering read and write operations, a memory-mapped interface provides access to disk storage via an array of bytes in main memory. The system call that maps a file into memory returns the virtual memory address that contains a copy of the file. The actual data transfers are performed only when needed to satisfy access to the memory image. Because the transfers are handled by the same mechanism as that used for demand-paged virtual memory access, memory-mapped I/O is efficient. Memory mapping is also convenient for programmers—access to a memory-mapped file is as simple as reading from and writing to memory. Operating systems that offer virtual memory commonly use the mapping interface for kernel services. For instance, to execute a program, the operating system maps the executable into memory and then transfers control to the entry address of the executable. The mapping interface is also commonly used for kernel access to swap space on disk.

A keyboard is an example of a device that is accessed through a **character-stream interface**. The basic system calls in this interface enable an application to `get()` or `put()` one character. On top of this interface, libraries can be built that offer line-at-a-time access, with buffering and editing services (for example, when a user types a backspace, the preceding character is removed from the input stream). This style of access is convenient for input devices such as keyboards, mice, and modems that produce data for input "spontaneously" —that is, at times that cannot necessarily be predicted by the application. This access style is also good for output devices such as printers and audio boards, which naturally fit the concept of a linear stream of bytes.

12.3.2 Network Devices

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

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

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

12.3.3 Clocks and Timers

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

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

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

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

Computers have clock hardware that is used for a variety of purposes. Modern PCs include a **high-performance event timer** (**HPET**), which runs at rates in the 10-megahertz range. It has several comparators that can be set to trigger once or repeatedly when the value they hold matches that of the HPET. The trigger generates an interrupt, and the operating system's clock management routines determine what the timer was for and what action to take. The precision of triggers is limited by the resolution of the timer, together with the overhead of maintaining virtual clocks. Furthermore, if the timer ticks are used to maintain the system time-of-day clock, the system clock can drift. Drift can be corrected via protocols designed for that purpose, such as NTP, the **network time protocol**, which uses sophisticated latency calculations to keep a computer's clock accurate almost to atomic-clock levels. In most computers, the hardware clock is constructed from a high-frequency counter. In some computers, the value of this counter can be read from a device register, in which case the counter can be considered a high-resolution clock. Although this clock does not generate interrupts, it offers accurate measurements of time intervals.

12.3.4 Nonblocking and Asynchronous I/O

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

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

One way an application writer can overlap execution with I/O is to write a multithreaded application. Some threads can perform blocking system calls, while others continue executing. Some operating systems provide nonblocking I/O system calls. A nonblocking call does not halt the execution of the thread for an extended time. Instead, it returns quickly, with a return value that indicates how many bytes were transferred.

An alternative to a nonblocking system call is an asynchronous system call. An asynchronous call returns immediately, without waiting for the I/O to complete. The thread continues to execute its code. The completion of the I/O at some future time is communicated to the thread, either through the setting of some variable in the address space of the thread or through the triggering of a signal or software interrupt or a call-back routine that is executed outside the linear control flow of the thread. The difference between nonblocking and asynchronous system calls is that a nonblocking `read()` returns immediately with whatever data are available—the full number of bytes requested, fewer, or none at all. An asynchronous `read()` call requests a transfer that will be performed in its entirety but will complete at some future time. These two I/O methods are shown in Figure 12.9.

Asynchronous activities occur throughout modern operating systems. Frequently, they are not exposed to users or applications but rather are contained within the operating-system operation. Secondary storage device and network I/O are useful examples. By default, when an application issues a network send request or a storage device write request, the operating system notes the request, buffers the I/O, and returns to the application. When possible, to optimize overall system performance, the operating system completes the request. If a system failure occurs in the interim, the application will lose any "in-flight" requests. Therefore, operating systems usually put a limit on how

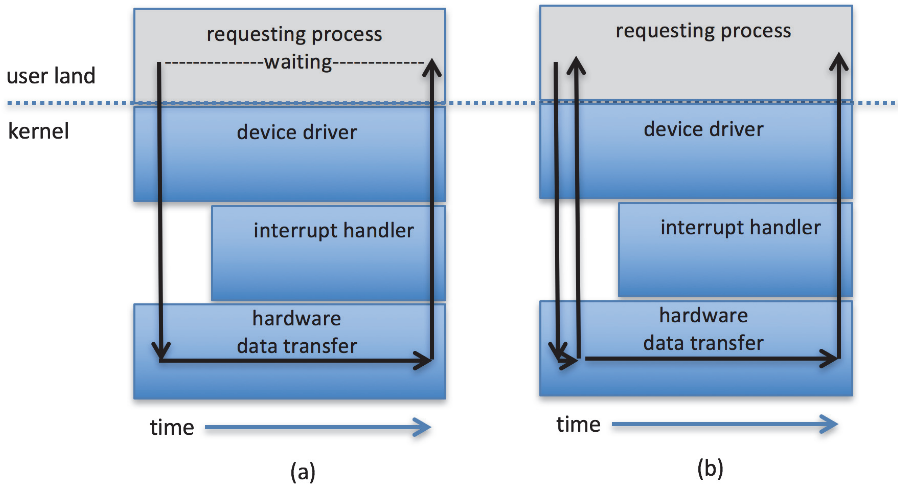


Figure 12.9 Two I/O methods: (a) synchronous and (b) asynchronous.

long they will buffer a request. Some versions of UNIX flush their secondary storage buffers every 30 seconds, for example, or each request is flushed within 30 seconds of its occurrence. Systems provide a way to allow applications to request a flush of some buffers (like secondary storage buffers) so the data can be forced to secondary storage without waiting for the buffer flush interval. Data consistency within applications is maintained by the kernel, which reads data from its buffers before issuing I/O requests to devices, ensuring that data not yet written are nevertheless returned to a requesting reader. Note that multiple threads performing I/O to the same file might not receive consistent data, depending on how the kernel implements its I/O. In this situation, the threads may need to use locking protocols. Some I/O requests need to be performed immediately, so I/O system calls usually have a way to indicate that a given request, or I/O to a specific device, should be performed synchronously.

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

12.3.5 Vectored I/O

Some operating systems provide another major variation of I/O via their application interfaces. **Vectored I/O** allows one system call to perform multiple I/O operations involving multiple locations. For example, the UNIX `readv` system call accepts a vector of multiple buffers and either reads from a source to that vector or writes from that vector to a destination. The same transfer could be caused by several individual invocations of system calls, but this **scatter–gather** method is useful for a variety of reasons.

Multiple separate buffers can have their contents transferred via one system call, avoiding context-switching and system-call overhead. Without vectored I/O, the data might first need to be transferred to a larger buffer in the right order and then transmitted, which is inefficient. In addition, some versions of scatter–gather provide atomicity, assuring that all the I/O is done without interruption (and avoiding corruption of data if other threads are also performing I/O involving those buffers). When possible, programmers make use of scatter–gather I/O features to increase throughput and decrease system overhead.

12.4 Kernel I/O Subsystem

Kernels provide many services related to I/O. Several services—scheduling, buffering, caching, spooling, device reservation, and error handling—are provided by the kernel's I/O subsystem and build on the hardware and device-driver infrastructure. The I/O subsystem is also responsible for protecting itself from errant processes and malicious users.

12.4.1 I/O Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete. Here is a simple example to illustrate. Suppose that a disk arm is near the beginning of a disk and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one near the beginning, and application 3 requests one in the middle of the disk. The operating system can reduce the distance that the disk arm travels by serving the applications in the order 2, 3, 1. Rearranging the order of service in this way is the essence of I/O scheduling.

Operating-system developers implement scheduling by maintaining a wait queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. The operating system may also try to be fair, so that no one application receives especially poor service, or it may give priority service for delay-sensitive requests. For instance, requests from the virtual memory subsystem may take priority over application requests. Several scheduling algorithms for disk I/O were detailed in Section 11.2.

When a kernel supports asynchronous I/O, it must be able to keep track of many I/O requests at the same time. For this purpose, the operating system might attach the wait queue to a **device-status table**. The kernel manages this table, which contains an entry for each I/O device, as shown in Figure 12.10. Each table entry indicates the device's type, address, and state (not functioning,

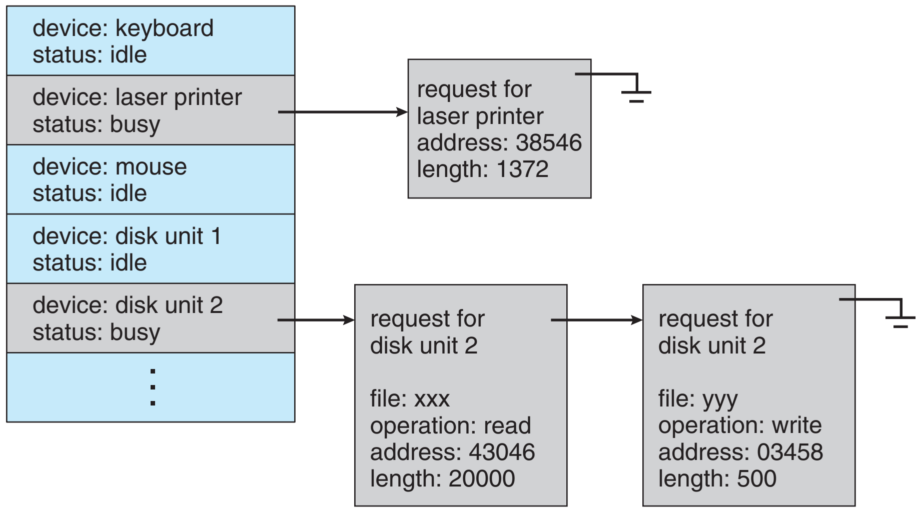


Figure 12.10 Device-status table.

idle, or busy). If the device is busy with a request, the type of request and other parameters will be stored in the table entry for that device.

Scheduling I/O operations is one way in which the I/O subsystem improves the efficiency of the computer. Another way is by using storage space in main memory or elsewhere in the storage hierarchy via buffering, caching, and spooling.

12.4.2 Buffering

A **buffer**, of course, is a memory area that stores data being transferred between two devices or between a device and an application. Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. Suppose, for example, that a file is being received via Internet for storage on an SSD. The network speed may be a thousand times slower than the drive. So a buffer is created in main memory to accumulate the bytes received from the network. When an entire buffer of data has arrived, the buffer can be written to the drive in a single operation. Since the drive write is not instantaneous and the network interface still needs a place to store additional incoming data, two buffers are used. After the network fills the first buffer, the drive write is requested. The network then starts to fill the second buffer while the first buffer is written to storage. By the time the network has filled the second buffer, the drive write from the first one should have completed, so the network can switch back to the first buffer while the drive writes the second one. This **double buffering** decouples the producer of data from the consumer, thus relaxing timing requirements between them. The need for this decoupling is illustrated in Figure 12.11, which lists the enormous differences in device speeds for typical computer hardware and interfaces.

A second use of buffering is to provide adaptations for devices that have different data-transfer sizes. Such disparities are especially common in computer networking, where buffers are used widely for fragmentation and

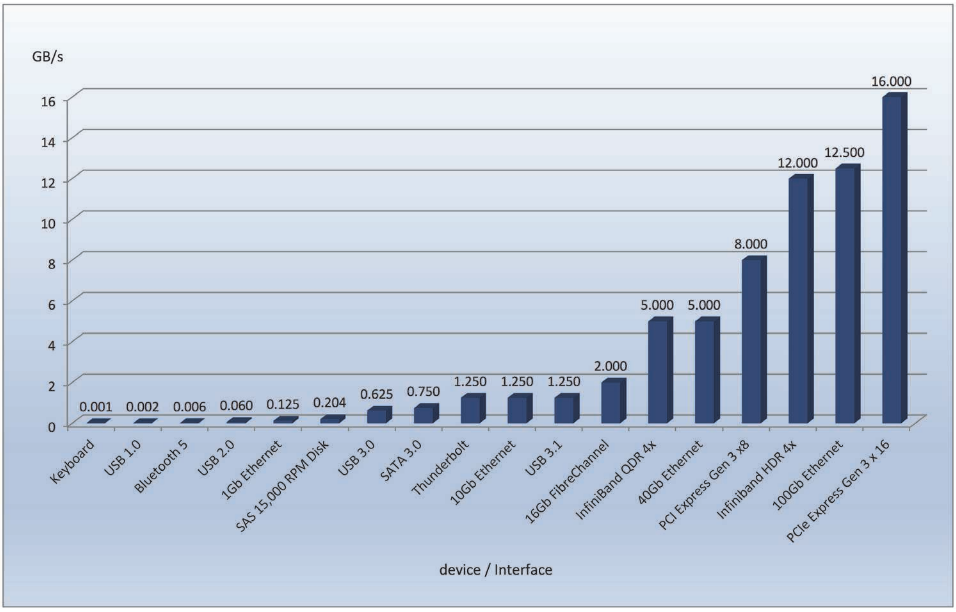


Figure 12.11 Common PC and data-center I/O device and interface speeds.

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

A third use of buffering is to support copy semantics for application I/O. An example will clarify the meaning of "copy semantics." Suppose that an application has a buffer of data that it wishes to write to disk. It calls the `write()` system call, providing a pointer to the buffer and an integer specifying the number of bytes to write. After the system call returns, what happens if the application changes the contents of the buffer? With **copy semantics**, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application's buffer. A simple way in which the operating system can guarantee copy semantics is for the `write()` system call to copy the application data into a kernel buffer before returning control to the application. The disk write is performed from the kernel buffer, so that subsequent changes to the application buffer have no effect. Copying of data between kernel buffers and application data space is common in operating systems, despite the overhead that this operation introduces, because of the clean semantics. The same effect can be obtained more efficiently by clever use of virtual memory mapping and copy-on-write page protection.

12.4.3 Caching

A **cache** is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches.

The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, holds a copy on faster storage of an item that resides elsewhere.

Caching and buffering are distinct functions, but sometimes a region of memory can be used for both purposes. For instance, to preserve copy semantics and to enable efficient scheduling of disk I/O, the operating system uses buffers in main memory to hold disk data. These buffers are also used as a cache, to improve the I/O efficiency for files that are shared by applications or that are being written and reread rapidly. When the kernel receives a file I/O request, the kernel first accesses the buffer cache to see whether that region of the file is already available in main memory. If it is, a physical disk I/O can be avoided or deferred. Also, disk writes are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules. This strategy of delaying writes to improve I/O efficiency is discussed, in the context of remote file access, in Section 19.8.

12.4.4 Spooling and Device Reservation

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

Some devices, such as tape drives and printers, cannot usefully multiplex the I/O requests of multiple concurrent applications. Spooling is one way operating systems can coordinate concurrent output. Another way to deal with concurrent device access is to provide explicit facilities for coordination. Some operating systems (including VMS) provide support for exclusive device access by enabling a process to allocate an idle device and to deallocate that device when it is no longer needed. Other operating systems enforce a limit of one open file handle to such a device. Many operating systems provide functions that enable processes to coordinate exclusive access among themselves. For instance, Windows provides system calls to wait until a device object becomes available. It also has a parameter to the `OpenFile()` system call that declares the types of access to be permitted to other concurrent threads. On these systems, it is up to the applications to avoid deadlock.

12.4.5 Error Handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical malfunction. Devices and I/O transfers can fail in many ways, either for transient reasons, as when a network becomes overloaded, or for “permanent” reasons, as when a disk controller becomes defective. Operating systems can often compensate effectively for transient failures. For instance, a disk `read()` failure results in a `read()` retry, and a network `send()` error results in a `resend()`, if the protocol so specifies. Unfortunately, if an important component experiences a permanent failure, the operating system is unlikely to recover.

As a general rule, an I/O system call will return one bit of information about the status of the call, signifying either success or failure. In the UNIX operating system, an additional integer variable named `errno` is used to return an error code—one of about a hundred values—indicating the general nature of the failure (for example, argument out of range, bad pointer, or file not open). By contrast, some hardware can provide highly detailed error information, although many current operating systems are not designed to convey this information to the application. For instance, a failure of a SCSI device is reported by the SCSI protocol in three levels of detail: a **sense key** that identifies the general nature of the failure, such as a hardware error or an illegal request; an **additional sense code** that states the category of failure, such as a bad command parameter or a self-test failure; and an **additional sense-code qualifie** that gives even more detail, such as which command parameter was in error or which hardware subsystem failed its self-test. Further, many SCSI devices maintain internal pages of error-log information that can be requested by the host—but seldom are.

12.4.6 I/O Protection

Errors are closely related to the issue of protection. A user process may accidentally or purposely attempt to disrupt the normal operation of a system by attempting to issue illegal I/O instructions. We can use various mechanisms to ensure that such disruptions cannot take place in the system.

To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system. To do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf (Figure 12.12). The operating system, executing in monitor mode, checks that the request is valid and, if it is, does the I/O requested. The operating system then returns to the user.

In addition, any memory-mapped and I/O port memory locations must be protected from user access by the memory-protection system. Note that a kernel cannot simply deny all user access. Most graphics games and video editing and playback software need direct access to memory-mapped graphics controller memory to speed the performance of the graphics, for example. The kernel might in this case provide a locking mechanism to allow a section of graphics memory (representing a window on screen) to be allocated to one process at a time.

12.4.7 Kernel Data Structures

The kernel needs to keep state information about the use of I/O components. It does so through a variety of in-kernel data structures, such as the open-file table

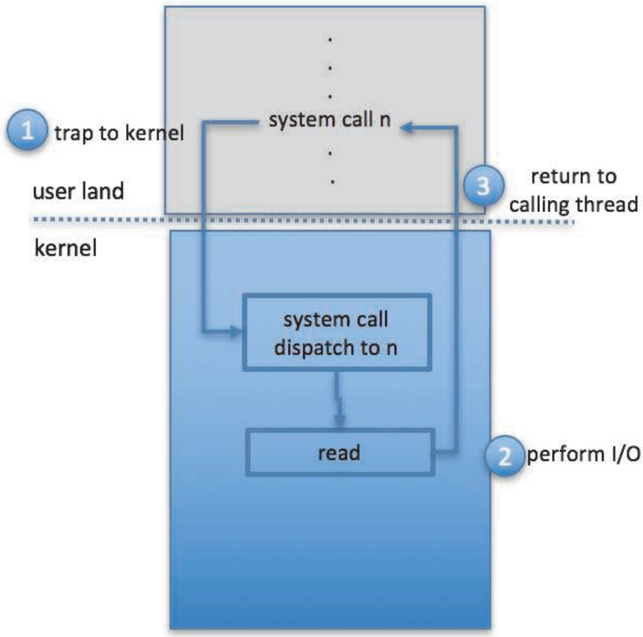


Figure 12.12 Use of a system call to perform I/O.

structure discussed in Section 14.1. The kernel uses many similar structures to track network connections, character-device communications, and other I/O activities.

UNIX provides file-system access to a variety of entities, such as user files, raw devices, and the address spaces of processes. Although each of these entities supports a `read()` operation, the semantics differ. For instance, to read a user file, the kernel needs to probe the buffer cache before deciding whether to perform a disk I/O. To read a raw disk, the kernel needs to ensure that the request size is a multiple of the disk sector size and is aligned on a sector boundary. To read a process image, it is merely necessary to copy data from memory. UNIX encapsulates these differences within a uniform structure by using an object-oriented technique. The open-file record, shown in Figure 12.13, contains a dispatch table that holds pointers to the appropriate routines, depending on the type of file.

Some operating systems use object-oriented methods even more extensively. For instance, Windows uses a message-passing implementation for I/O. An I/O request is converted into a message that is sent through the kernel to the I/O manager and then to the device driver, each of which may change the message contents. For output, the message contains the data to be written. For input, the message contains a buffer to receive the data. The message-passing approach can add overhead, by comparison with procedural techniques that use shared data structures, but it simplifies the structure and design of the I/O system and adds flexibility.

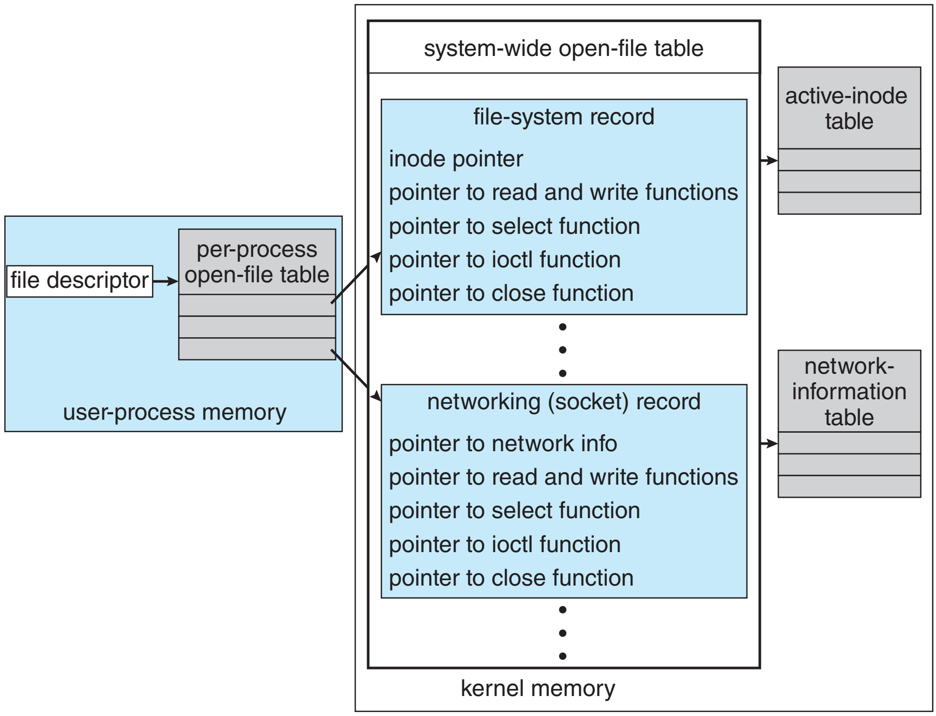


Figure 12.13 UNIX I/O kernel structure.

12.4.8 Power Management

Computers residing in data centers may seem far removed from issues of power use, but as power costs increase and the world becomes increasingly troubled about the long-term effects of greenhouse gas emissions, data centers have become a cause for concern and a target for increased efficiencies. Electricity use generates heat, and computer components can fail due to high temperatures, so cooling is part of the equation as well. Consider that cooling a modern data center may use twice as much electricity as powering the equipment does. Many approaches to data-center power optimization are in use, ranging from interchanging data-center air without side air, chilling with natural sources such as lake water, and solar panels.

Operating systems play a role in power use (and therefore heat generation and cooling). In cloud computing environments, processing loads can be adjusted by monitoring and management tools to evacuate all user processes from systems, idling those systems and powering them off until the load requires their use. An operating system could analyze its load and, if sufficiently low and hardware-enabled, power off components such as CPUsand external I/O devices.

CPU cores can be suspended when the system load does not require them and resumed when the load increases and more cores are needed to run the queue of threads. Their state, of course, needs to be saved on suspend and restored on resume. This feature is needed in servers because a data center full of servers can use vast amounts of electricity, and disabling unneeded cores can decrease electricity (and cooling) needs.

In mobile computing, power management becomes a high-priority aspect of the operating system. Minimizing power use and therefore maximizing battery life increases the usability of a device and helps it compete with alternative devices. Today's mobile devices offer the functionality of yesterday's high-end desktop, yet are powered by batteries and are small enough to fit in your pocket. In order to provide satisfactory battery life, modern mobile operating systems are designed from the ground up with power management as a key feature. Let's examine in detail three major features that enable the popular Android mobile system to maximize battery life: power collapse, component-level power management, and wakelocks.

Power collapse is the ability to put a device into a very deep sleep state. The device uses only marginally more power than if it were fully powered off, yet it is still able to respond to external stimuli, such as the user pressing a button, at which time it quickly powers back on. Power collapse is achieved by powering off many of the individual components within a device—such as the screen, speakers, and I/O subsystem—so that they consume no power. The operating system then places the CPU in its lowest sleep state. A modern ARM CPU might consume hundreds of milliwatts per core under typical load yet only a handful of milliwatts in its lowest sleep state. In such a state, although the CPU is idle, it can receive an interrupt, wake up, and resume its previous activity very quickly. Thus, an idle Android phone in your pocket uses very little power, but it can spring to life when it receives a phone call.

How is Android able to turn off the individual components of a phone? How does it know when it is safe to power off the flash storage, and how does it know to do that before powering down the overall I/O subsystem? The answer is component-level power management, which is an infrastructure that understands the relationship between components and whether each component is in use. To understand the relationship between components, Android builds a device tree that represents the phone's physical-device topology. For example, in such a topology, flash and USB storage would be sub-nodes of the I/O subsystem, which is a sub-node of the system bus, which in turn connects to the CPU. To understand usage, each component is associated with its device driver, and the driver tracks whether the component is in use—for example, if there is I/O pending to flash or if an application has an open reference to the audio subsystem. With this information, Android can manage the power of the phone's individual components: If a component is unused, it is turned off. If all of the components on, say, the system bus are unused, the system bus is turned off. And if all of the components in the entire device tree are unused, the system may enter power collapse.

With these technologies, Android can aggressively manage its power consumption. But a final piece of the solution is missing: the ability for applications to temporarily prevent the system from entering power collapse. Consider a user playing a game, watching a video, or waiting for a web page to open. In all of these cases, the application needs a way to keep the device awake, at least temporarily. Wakelocks enable this functionality. Applications acquire and release wakelocks as needed. While an application holds a wakelock, the kernel will prevent the system from entering power collapse. For example, while the Android Market is updating an application, it will hold a wakelock to ensure that the system does not go to sleep until the update is complete. Once complete, the Android Market will release the wakelock, allowing the system to enter power collapse.

Power management in general is based on device management, which is more complicated than we have so far portrayed it. At boot time, the firmware system analyzes the system hardware and creates a device tree in RAM. The kernel then uses that device tree to load device drivers and manage devices. Many additional activities pertaining to devices must be managed, though, including addition and subtraction of devices from a running system ("hot-plug"), understanding and changing device states, and power management. Modern general-purpose computers use another set of firmware code, **advanced configuratio and power interface** (**ACPI**), to manage these aspects of hardware. ACPI is an industry standard (http://www.acpi.info) with many features. It provides code that runs as routines callable by the kernel for device state discovery and management, device error management, and power management. For example, when the kernel needs to quiesce a device, it calls the device driver, which calls the ACPI routines, which then talk to the device.

12.4.9 Kernel I/O Subsystem Summary

In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel. The I/O subsystem supervises these procedures:

- Management of the name space for files and devices
- Access control to files and devices
- Operation control (for example, a modem cannot `seek()`)
- File-system space allocation
- Device allocation
- Buffering, caching, and spooling
- I/O scheduling
- Device-status monitoring, error handling, and failure recovery
- Device-driver configuration and initialization
- Power management of I/O devices

The upper levels of the I/O subsystem access devices via the uniform interface provided by the device drivers.

12.5 Transforming I/O Requests to Hardware Operations

Earlier, we described the handshaking between a device driver and a device controller, but we did not explain how the operating system connects an application request to a set of network wires or to a specific disk sector. Consider, for example, reading a file from disk. The application refers to the data by a file name. Within a disk, the file system maps from the file name through the file-system directories to obtain the space allocation of the file. For instance, in MS-DOS for FAT (a relatively simple operating and file system still used today as a common interchange format), the name maps to a number that indicates an entry in the file-access table, and that table entry tells which disk blocks are allocated to the file. In UNIX, the name maps to an inode number, and the corresponding inode contains the space-allocation information. But how is the connection made from the file name to the disk controller (the hardware port address or the memory-mapped controller registers)?

One method is that used by MS-DOS for FAT, mentioned above. The first part of an MS-DOS file name, preceding the colon, is a string that identifies a specific hardware device. For example, C: is the first part of every file name on the primary hard disk. The fact that C: represents the primary hard disk is built into the operating system; C: is mapped to a specific port address through a device table. Because of the colon separator, the device name space is separate from the file-system name space. This separation makes it easy for the operating system to associate extra functionality with each device. For instance, it is easy to invoke spooling on any files written to the printer.

If, instead, the device name space is incorporated in the regular file-system name space, as it is in UNIX, the normal file-system name services are provided automatically. If the file system provides ownership and access control to all file names, then devices have owners and access control. Since files are stored on devices, such an interface provides access to the I/O system at two levels. Names can be used to access the devices themselves or to access the files stored on the devices.

UNIX represents device names in the regular file-system name space. Unlike an MS-DOS FAT file name, which has a colon separator, a UNIX path name has no clear separation of the device portion. In fact, no part of the path name is the name of a device. UNIX has a **mount table** that associates prefixes of path names with specific device names. To resolve a path name, UNIX looks up the name in the mount table to find the longest matching prefix; the corresponding entry in the mount table gives the device name. This device name also has the form of a name in the file-system name space. When UNIX looks up this name in the file-system directory structures, it finds not an inode number but a <major, minor> device number. The major device number identifies a device driver that should be called to handle I/O to this device. The minor device number is passed to the device driver to index into a device table. The corresponding device-table entry gives the port address or the memory-mapped address of the device controller.

Modern operating systems gain significant flexibility from the multiple stages of lookup tables in the path between a request and a physical device controller. The mechanisms that pass requests between applications and drivers are general. Thus, we can introduce new devices and drivers into a computer without recompiling the kernel. In fact, some operating systems have the ability to load device drivers on demand. At boot time, the system first probes the hardware buses to determine what devices are present. It then loads the necessary drivers, either immediately or when first required by an I/O request. Devices added after boot can be detected by the error they cause (interrupt-generated with no associated interrupt handler, for example), which can prompt the kernel to inspect the device details and load an appropriate device driver dynamically. Of course, dynamic loading (and unloading) is more complicated than static loading, requiring more complex kernel algorithms, device-structure locking, error handling, and so forth.

We next describe the typical life cycle of a blocking read request, as depicted in Figure 12.14. The figure suggests that an I/O operation requires a great many steps that together consume a tremendous number of CPU cycles.

1. A process issues a blocking `read()` system call to a file descriptor of a file that has been opened previously.
2. The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is completed.
3. Otherwise, a physical I/O must be performed. The process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.

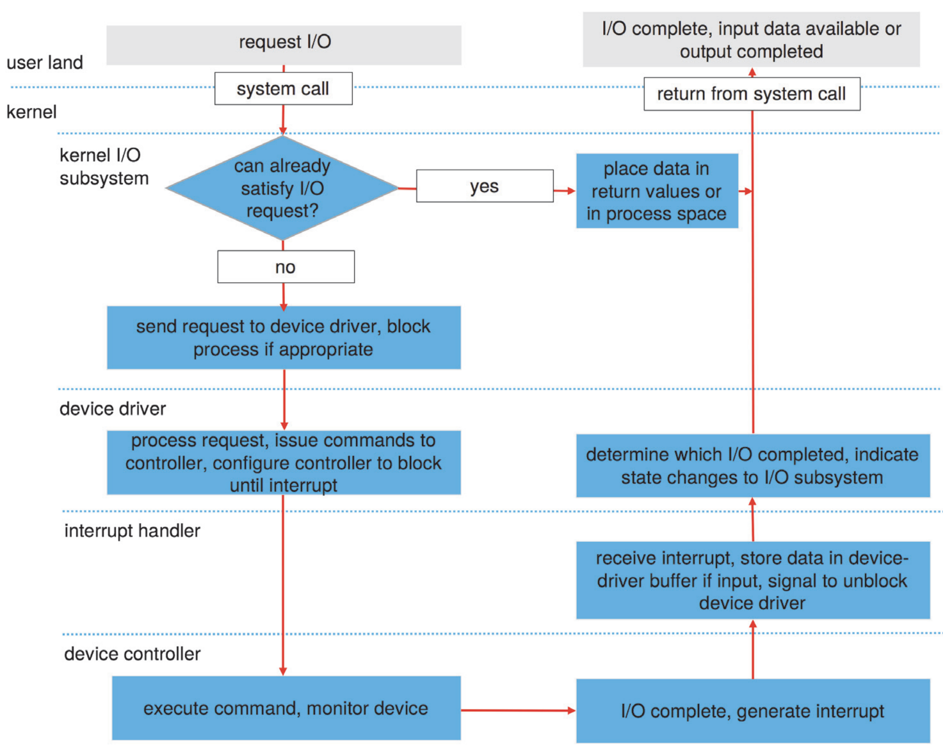


Figure 12.14 The life cycle of an I/O request.

4. The device driver allocates kernel buffer space to receive the data and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device-control registers.
5. The device controller operates the device hardware to perform the data transfer.
6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.
7. The correct interrupt handler receives the interrupt via the interrupt-vector table, stores any necessary data, signals the device driver, and returns from the interrupt.
8. The device driver receives the signal, determines which I/O request has completed, determines the request's status, and signals the kernel I/O subsystem that the request has been completed.
9. The kernel transfers data or return codes to the address space of the requesting process and moves the process from the wait queue back to the ready queue.
10. Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.

12.6 STREAMS

UNIX System V (and many subsequent UNIX releases) has an interesting mechanism, called **STREAMS**, that enables an application to assemble pipelines of driver code dynamically. A stream is a full-duplex connection between a device driver and a user-level process. It consists of a **stream head** that interfaces with the user process, a **driver end** that controls the device, and zero or more **stream modules** between the stream head and the driver end. Each of these components contains a pair of queues—a read queue and a write queue. Message passing is used to transfer data between queues. The STREAMS structure is shown in Figure 12.15.

Modules provide the functionality of STREAMS processing; they are ***pushed*** onto a stream by use of the `ioctl()` system call. For example, a process can open a USB device (like a keyboard) via a stream and can push on a module to handle input editing. Because messages are exchanged between queues in adjacent modules, a queue in one module may overflow an adjacent queue. To prevent this from occurring, a queue may support **flo control**. Without flow control, a queue accepts all messages and immediately sends them on to the queue in the adjacent module without buffering them. A queue that supports flow control buffers messages and does not accept messages without sufficient buffer space. This process involves exchanges of control messages between queues in adjacent modules.

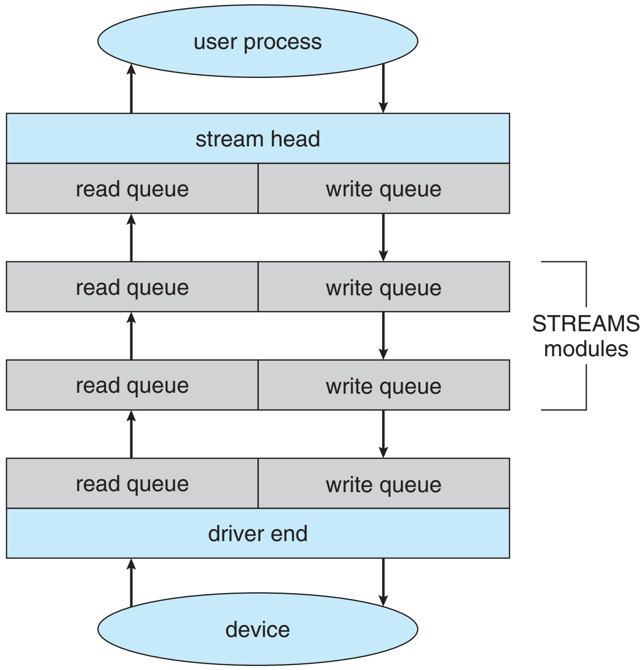


Figure 12.15 The STREAMS structure.

A user process writes data to a device using either the `write()` or `putmsg()` system call. The `write()` system call writes raw data to the stream, whereas `putmsg()` allows the user process to specify a message. Regardless of the system call used by the user process, the stream head copies the data into a message and delivers it to the queue for the next module in line. This copying of messages continues until the message is copied to the driver end and hence the device. Similarly, the user process reads data from the stream head using either the `read()` or `getmsg()` system call. If `read()` is used, the stream head gets a message from its adjacent queue and returns ordinary data (an unstructured byte stream) to the process. If `getmsg()` is used, a message is returned to the process.

STREAMS I/O is asynchronous (or nonblocking) except when the user process communicates with the stream head. When writing to the stream, the user process will block, assuming the next queue uses flow control, until there is room to copy the message. Likewise, the user process will block when reading from the stream until data are available.

As mentioned, the driver end—like the stream head and modules—has a read and write queue. However, the driver end must respond to interrupts, such as one triggered when a frame is ready to be read from a network. Unlike the stream head, which may block if it is unable to copy a message to the next queue in line, the driver end must handle all incoming data. Drivers must support flow control as well. However, if a device's buffer is full, the device typically resorts to dropping incoming messages. Consider a network card whose input buffer is full. The network card must simply drop further messages until there is enough buffer space to store incoming messages.

The benefit of using STREAMS is that it provides a framework for a modular and incremental approach to writing device drivers and network protocols. Modules may be used by different streams and hence by different devices. For example, a networking module may be used by both an Ethernet network card and a 802.11 wireless network card. Furthermore, rather than treating character-device I/O as an unstructured byte stream, STREAMS allows support for message boundaries and control information when communicating between modules. Most UNIX variants support STREAMS, and it is the preferred method for writing protocols and device drivers. For example, System V UNIX and Solaris implement the socket mechanism using STREAMS.

12.7 Performance

I/O is a major factor in system performance. It places heavy demands on the CPU to execute device-driver code and to schedule processes fairly and efficiently as they block and unblock. The resulting context switches stress the CPU and its hardware caches. I/O also exposes any inefficiencies in the interrupt-handling mechanisms in the kernel. In addition, I/O loads down the memory bus during data copies between controllers and physical memory and again during copies between kernel buffers and application data space. Coping gracefully with all these demands is one of the major concerns of a computer architect.

Although modern computers can handle many thousands of interrupts per second, interrupt handling is a relatively expensive task. Each interrupt causes the system to perform a state change, to execute the interrupt handler, and then to restore state. Programmed I/O can be more efficient than interrupt-driven I/O, if the number of cycles spent in busy waiting is not excessive. An I/O completion typically unblocks a process, leading to the full overhead of a context switch.

Network traffic can also cause a high context-switch rate. Consider, for instance, a remote login from one machine to another. Each character typed on the local machine must be transported to the remote machine. On the local machine, the character is typed; a keyboard interrupt is generated; and the character is passed through the interrupt handler to the device driver, to the kernel, and then to the user process. The user process issues a network I/O system call to send the character to the remote machine. The character then flows into the local kernel, through the network layers that construct a network packet, and into the network device driver. The network device driver transfers the packet to the network controller, which sends the character and generates an interrupt. The interrupt is passed back up through the kernel to cause the network I/O system call to complete.

Now, the remote system's network hardware receives the packet, and an interrupt is generated. The character is unpacked from the network protocols and is given to the appropriate network daemon. The network daemon identifies which remote login session is involved and passes the packet to the appropriate subdaemon for that session. Throughout this flow, there are context switches and state switches (Figure 12.16). Usually, the receiver echoes the character back to the sender; that approach doubles the work.

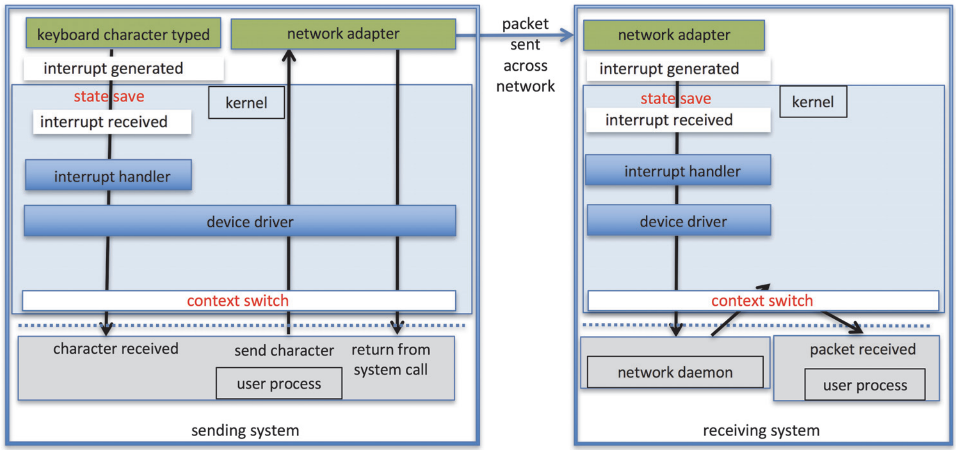


Figure 12.16 Intercomputer communications.

Some systems use separate **front-end processors** for terminal I/O to reduce the interrupt burden on the main CPU. For instance, a **terminal concentrator** can multiplex the traffic from hundreds of remote terminals into one port on a large computer. An **I/O channel** is a dedicated, special-purpose CPU found in mainframes and in other high-end systems. The job of a channel is to offload I/O work from the main CPU. The idea is that the channels keep the data flowing smoothly, while the main CPU remains free to process the data. Like the device controllers and DMA controllers found in smaller computers, a channel can process more general and sophisticated programs, so channels can be tuned for particular workloads.

We can employ several principles to improve the efficiency of I/O:

- Reduce the number of context switches.
- Reduce the number of times that data must be copied in memory while passing between device and application.
- Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy waiting can be minimized).
- Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
- Move processing primitives into hardware, to allow their operation in device controllers to be concurrent with CPU and bus operation.
- Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.

I/O devices vary greatly in complexity. For instance, a mouse is simple. The mouse movements and button clicks are converted into numeric values that are passed from hardware, through the mouse device driver, to the application. By contrast, the functionality provided by the Windows disk device driver is complex. It not only manages individual disks but also implements RAID arrays

(Section 11.8). To do so, it converts an application's read or write request into a coordinated set of disk I/O operations. Moreover, it implements sophisticated error-handling and data-recovery algorithms and takes many steps to optimize disk performance.

Where should the I/O functionality be implemented—in the device hardware, in the device driver, or in application software? Sometimes we observe the progression depicted in Figure 12.17.

- Initially, we implement experimental I/O algorithms at the application level, because application code is flexible and application bugs are unlikely to cause system crashes. Furthermore, by developing code at the application level, we avoid the need to reboot or reload device drivers after every change to the code. An application-level implementation can be inefficient, however, because of the overhead of context switches and because the application cannot take advantage of internal kernel data structures and kernel functionality (such as efficient in-kernel messaging, threading, and locking). The FUSE system interface, for example, allows file systems to be written and run in user mode.
- When an application-level algorithm has demonstrated its worth, we may reimplement it in the kernel. This can improve performance, but the development effort is more challenging, because an operating-system kernel is a large, complex software system. Moreover, an in-kernel implementation must be thoroughly debugged to avoid data corruption and system crashes.
- The highest performance may be obtained through a specialized implementation in hardware, either in the device or in the controller. The disadvantages of a hardware implementation include the difficulty and expense of making further improvements or of fixing bugs, the increased devel-

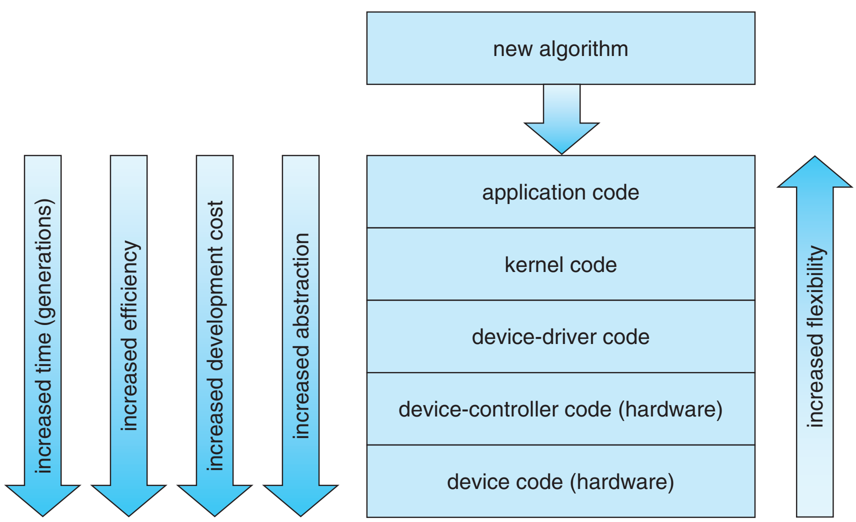


Figure 12.17 Device functionality progression.

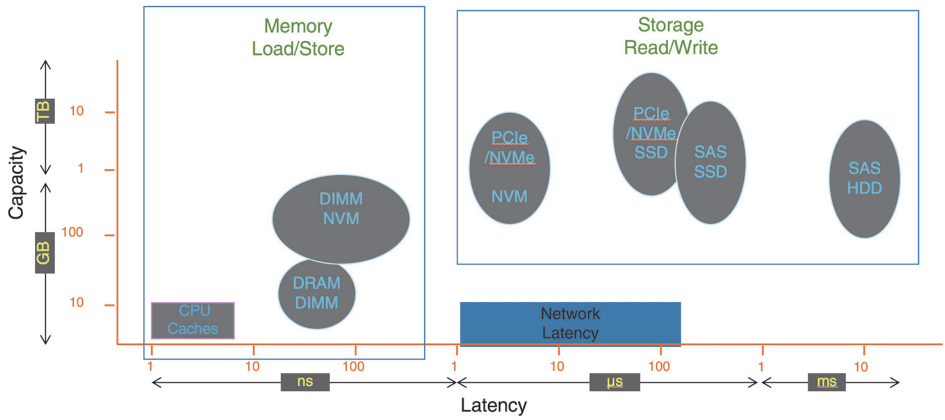


Figure 12.18 I/O performance of storage (and network latency).

opment time (months rather than days), and the decreased flexibility. For instance, a hardware RAID controller may not provide any means for the kernel to influence the order or location of individual block reads and writes, even if the kernel has special information about the workload that would enable it to improve the I/O performance.

Over time, as with other aspects of computing, I/O devices have been increasing in speed. Nonvolatile memory devices are growing in popularity and in the variety of devices available. The speed of NVM devices varies from high to extraordinary, with next-generation devices nearing the speed of DRAM. These developments are increasing pressure on I/O subsystems as well as operating system algorithms to take advantage of the read/write speeds now available. Figure 12.18 shows CPU and storage devices in two dimensions: capacity and latency of I/O operations. Added to the figure is a representation of networking latency to reveal the performance "tax" networking adds to I/O.

12.8 Summary

- The basic hardware elements involved in I/O are buses, device controllers, and the devices themselves.
- The work of moving data between devices and main memory is performed by the CPU as programmed I/O or is offloaded to a DMA controller.
- The kernel module that controls a device is a device driver. The system-call interface provided to applications is designed to handle several basic categories of hardware, including block devices, character-stream devices, memory-mapped files, network sockets, and programmed interval timers. The system calls usually block the processes that issue them, but nonblocking and asynchronous calls are used by the kernel itself and by applications that must not sleep while waiting for an I/O operation to complete.

- The kernel's I/O subsystem provides numerous services. Among these are I/O scheduling, buffering, caching, spooling, device reservation, error handling. Another service, name translation, makes the connections between hardware devices and the symbolic file names used by applications. It involves several levels of mapping that translate from character-string names, to specific device drivers and device addresses, and then to physical addresses of I/O ports or bus controllers. This mapping may occur within the file-system name space, as it does in UNIX, or in a separate device name space, as it does in MS-DOS.
- STREAMS is an implementation and methodology that provides a framework for a modular and incremental approach to writing device drivers and network protocols. Through STREAMS, drivers can be stacked, with data passing through them sequentially and bidirectionally for processing.
- I/O system calls are costly in terms of CPU consumption because of the many layers of software between a physical device and an application. These layers imply overhead from several sources: context switching to cross the kernel's protection boundary, signal and interrupt handling to service the I/O devices, and the load on the CPU and memory system to copy data between kernel buffers and application space.

Practice Exercises

12.1 State three advantages of placing functionality in a device controller, rather than in the kernel. State three disadvantages.

12.2 The example of handshaking in Section 12.2 used two bits: a busy bit and a command-ready bit. Is it possible to implement this handshaking with only one bit? If it is, describe the protocol. If it is not, explain why one bit is insufficient.

12.3 Why might a system use interrupt-driven I/O to manage a single serial port and polling I/O to manage a front-end processor, such as a terminal concentrator?

12.4 Polling for an I/O completion can waste a large number of CPU cycles if the processor iterates a busy-waiting loop many times before the I/O completes. But if the I/O device is ready for service, polling can be much more efficient than catching and dispatching an interrupt. Describe a hybrid strategy that combines polling, sleeping, and interrupts for I/O device service. For each of these three strategies (pure polling, pure interrupts, hybrid), describe a computing environment in which that strategy is more efficient than either of the others.

12.5 How does DMA increase system concurrency? How does it complicate hardware design?

12.6 Why is it important to scale up system-bus and device speeds as CPU speed increases?

12.7 Distinguish between a driver end and a stream module in a STREAMS operation.

Further Reading

[Hennessy and Patterson (2012)] describe multiprocessor systems and cache-consistency issues. [Intel (2011)] is a good source of information for Intel processors.

Details about PCIe can be found at https://pcisig.com. For more about ACPI see http://www.acpi.info.

The use of FUSE for user-mode file systems can create performance problems. An analysis of those issues can be found in https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor.

Bibliography

[Hennessy and Patterson (2012)] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Fifth Edition, Morgan Kaufmann (2012).

[Intel (2011)] *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 3A and 3B*. Intel Corporation (2011).

Chapter 12 Exercises

12.8 When multiple interrupts from different devices appear at about the same time, a priority scheme could be used to determine the order in which the interrupts would be serviced. Discuss what issues need to be considered in assigning priorities to different interrupts.

12.9 What are the advantages and disadvantages of supporting memory-mapped I/O to device-control registers?

12.10 Consider the following I/O scenarios on a single-user PC:

a. A mouse used with a graphical user interface

b. A tape drive on a multitasking operating system (with no device preallocation available)

c. A disk drive containing user files

d. A graphics card with direct bus connection, accessible through memory-mapped I/O

For each of these scenarios, would you design the operating system to use buffering, spooling, caching, or a combination? Would you use polled I/O or interrupt-driven I/O? Give reasons for your choices.

12.11 In most multiprogrammed systems, user programs access memory through virtual addresses, while the operating system uses raw physical addresses to access memory. What are the implications of this design for the initiation of I/O operations by the user program and their execution by the operating system?

12.12 What are the various kinds of performance overhead associated with servicing an interrupt?

12.13 Describe three circumstances under which blocking I/O should be used. Describe three circumstances under which nonblocking I/O should be used. Why not just implement nonblocking I/O and have processes busy-wait until their devices are ready?

12.14 Typically, at the completion of a device I/O, a single interrupt is raised and appropriately handled by the host processor. In certain settings, however, the code that is to be executed at the completion of the I/O can be broken into two separate pieces. The first piece executes immediately after the I/O completes and schedules a second interrupt for the remaining piece of code to be executed at a later time. What is the purpose of using this strategy in the design of interrupt handlers?

12.15 Some DMA controllers support direct virtual memory access, where the targets of I/O operations are specified as virtual addresses and a translation from virtual to physical address is performed during the DMA. How does this design complicate the design of the DMA controller? What are the advantages of providing such functionality?

12.16 UNIX coordinates the activities of the kernel I/O components by manipulating shared in-kernel data structures, whereas Windows uses object-oriented message passing between kernel I/O components. Discuss three pros and three cons of each approach.

12.17 Write (in pseudocode) an implementation of virtual clocks, including the queueing and management of timer requests for the kernel and applications. Assume that the hardware provides three timer channels.

12.18 Discuss the advantages and disadvantages of guaranteeing reliable transfer of data between modules in the STREAMS abstraction.

Part Six

File System

A *file* is a collection of related information defined by its creator. Files are mapped by the operating system onto physical mass-storage devices. A file system describes how files are mapped onto physical devices, as well as how they are accessed and manipulated by both users and programs.

Accessing physical storage can often be slow, so file systems must be designed for efficient access. Other requirements may be important as well, including providing support for file sharing and remote access to files.

CHAPTER 13

File-System Interface

For most users, the file system is the most visible aspect of a general-purpose operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system. Most file systems live on storage devices, which we described in Chapter 11 and will continue to discuss in the next chapter. In this chapter, we consider the various aspects of files and the major directory structures. We also discuss the semantics of sharing files among multiple processes, users, and computers. Finally, we discuss ways to handle file protection, necessary when we have multiple users and want to control who may access files and how files may be accessed.

CHAPTER OBJECTIVES

- Explain the function of file systems.
- Describe the interfaces to file systems.
- Discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures.
- Explore file-system protection.

13.1 File Concept

Computers can store information on various storage media, such as NVM devices, HDDs, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of stored information. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the **fil** . Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent between system reboots.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

Because files are **the** method users and applications use to store and retrieve data, and because they are so general purpose, their use has stretched beyond its original confines. For example, UNIX, Linux, and some other operating systems provide a `proc` file system that uses file-system interfaces to provide access to system information (such as process details).

The information in a file is defined by its creator. Many different types of information may be stored in a file—source or executable programs, numeric or text data, photos, music, video, and so on. A file has a certain defined structure, which depends on its type. A **text fil** is a sequence of characters organized into lines (and possibly pages). A **source fil** is a sequence of functions, each of which is further organized as declarations followed by executable statements. An **executable fil** is a series of code sections that the loader can bring into memory and execute.

13.1.1 File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as `example.c`. Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file `example.c`, and another user might edit that file by specifying its name. The file's owner might write the file to a USB drive, send it as an e-mail attachment, or copy it across a network, and it could still be called `example.c` on the destination system. Unless there is a sharing and synchonization method, that second copy is now independent of the first and can be changed separately.

A file's attributes vary from one operating system to another but typically consist of these:

- **Name**. The symbolic file name is the only information kept in human-readable form.
- **Identifie** . This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type**. This information is needed for systems that support different types of files.
- **Location**. This information is a pointer to a device and to the location of the file on that device.

- **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection**. Access-control information determines who can do reading, writing, executing, and so on.
- **Timestamps and user identificatio** . This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

Some newer file systems also support **extended file attributes**, including character encoding of the file and security features such as a file checksum. Figure 13.1 illustrates a **fil info window** on macOS that displays a file's attributes.

The information about all files is kept in the directory structure, which resides on the same device as the files themselves. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes. It may take more than a kilobyte to record this information for each file. In a system with many files, the size of the directory itself may be megabytes or gigabytes. Because directories must match the volatility of the files, like files, they must be stored on the device and are usually brought into memory piecemeal, as needed.

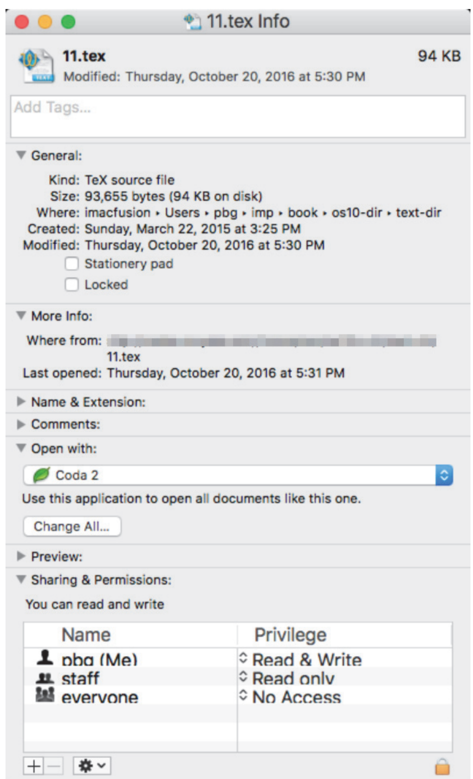


Figure 13.1 A file info window on macOS.

13.1.2 File Operations

A file is an abstract data type. To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these seven basic file operations. It should then be easy to see how other similar operations, such as renaming a file, can be implemented.

- **Creating a fil** . Two steps are necessary to create a file. First, space in the file system must be found for the file. We discuss how to allocate space for the file in Chapter 14. Second, an entry for the new file must be made in a directory.
- **Opening a fil** . Rather than have all file operations specify a file name, causing the operating system to evaluate the name, check access permissions, and so on, all operations except create and delete require a file `open()` first. If successful, the open call returns a file handle that is used as an argument in the other calls.
- **Writing a fil** . To write a file, we make a system call specifying both the open file handle and the information to be written to the file. The system must keep a **write pointer** to the location in the file where the next write is to take place if it is sequential. The write pointer must be updated whenever a write occurs.
- **Reading a fil** . To read from a file, we use a system call that specifies the file handle and where (in memory) the next block of the file should be put. Again, the system needs to keep a **read pointer** to the location in the file where the next read is to take place, if sequential. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **current-file-positio pointer**. Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **Repositioning within a file**. The current-file-position pointer of the open file is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file **seek**.
- **Deleting a fil** . To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase or mark as free the directory entry. Note that some systems allow **hard links**—multiple names (directory entries) for the same file. In this case the actual file contents is not deleted until the last link is deleted.
- **Truncating a fil** . The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length. The file can then be reset to length zero, and its file space can be released.

These seven basic operations comprise the minimal set of required file operations. Other common operations include appending new information to the end of an existing file and renaming an existing file. These primitive operations can then be combined to perform other file operations. For instance, we can create a copy of a file by creating a new file and then reading from the old and writing to the new. We also want to have operations that allow a user to get and set the various attributes of a file. For example, we may want to have operations that allow a user to determine the status of a file, such as the file's length, and to set file attributes, such as the file's owner.

As mentioned, most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an `open()` system call be made before a file is first used. The operating system keeps a table, called the **open-fil table**, containing information about all open files. When a file operation is requested, the file is specified via an index into this table, so no searching is required. When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table, potentially releasing locks. `create()` and `delete()` are system calls that work with closed rather than open files.

Some systems implicitly open a file when the first reference to it is made. The file is automatically closed when the job or program that opened the file terminates. Most systems, however, require that the programmer open a file explicitly with the `open()` system call before that file can be used. The `open()` operation takes a file name and searches the directory, copying the directory entry into the open-file table. The `open()` call can also accept access-mode information—create, read-only, read–write, append-only, and so on. This mode is checked against the file's permissions. If the request mode is allowed, the file is opened for the process. The `open()` system call typically returns a pointer to the entry in the open-file table. This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching and simplifying the system-call interface.

The implementation of the `open()` and `close()` operations is more complicated in an environment where several processes may open the file simultaneously. This may occur in a system where several different applications open the same file at the same time. Typically, the operating system uses two levels of internal tables: a per-process table and a system-wide table. The per-process table tracks all files that a process has open. Stored in this table is information regarding the process's use of the file. For instance, the current file pointer for each file is found here. Access rights to the file and accounting information can also be included.

Each entry in the per-process table in turn points to a system-wide open-file table. The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an `open()` call, a new entry is simply added to the process's open-file table pointing to the appropriate entry in the system-wide table. Typically, the open-file table also has an **open count** associated with each file to indicate how many processes have the file open. Each `close()` decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.

FILE LOCKING IN JAVA

In the Java API, acquiring a lock requires first obtaining the `FileChannel` for the file to be locked. The `lock()` method of the `FileChannel` is used to acquire the lock. The API of the `lock()` method is

```
FileLock lock(long begin, long end, boolean shared)
```

where `begin` and `end` are the beginning and ending positions of the region being locked. Setting `shared` to `true` is for shared locks; setting `shared` to `false` acquires the lock exclusively. The lock is released by invoking the `release()` of the `FileLock` returned by the `lock()` operation.

The program in Figure 13.2 illustrates file locking in Java. This program acquires two locks on the file `file.txt`. The lock for the first half of the file is an exclusive lock; the lock for the second half is a shared lock.

In summary, several pieces of information are associated with an open file.

- **File pointer**. On systems that do not include a file offset as part of the `read()` and `write()` system calls, the system must track the last read–write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
- **File-open count**. As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Multiple processes may have opened a file, and the system must wait for the last file to close before removing the open-file table entry. The file-open count tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.
- **Location of the fil** . Most file operations require the system to read or write data within the file. The information needed to locate the file (wherever it is located, be it on mass storage, on a file server across the network, or on a RAM drive) is kept in memory so that the system does not have to read it from the directory structure for each operation.
- **Access rights**. Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

Some operating systems provide facilities for locking an open file (or sections of a file). File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes—for example, a system log file that can be accessed and modified by a number of processes in the system.

File locks provide functionality similar to reader–writer locks, covered in Section 7.1.2. A **shared lock** is akin to a reader lock in that several processes can acquire the lock concurrently. An **exclusive lock** behaves like a writer lock; only one process at a time can acquire such a lock. It is important to note that not

```
import java.io.*;
import java.nio.channels.*;

public class LockingExample {
 public static final boolean EXCLUSIVE = false;
 public static final boolean SHARED = true;

 public static void main(String args[]) throws IOException {
  FileLock sharedLock = null;
  FileLock exclusiveLock = null;

  try {
   RandomAccessFile raf = new RandomAccessFile("file.txt","rw");

   // get the channel for the file
   FileChannel ch = raf.getChannel();

   // this locks the first half of the file - exclusive
   exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);

   /** Now modify the data . . . */

   // release the lock
   exclusiveLock.release();

   // this locks the second half of the file - shared
   sharedLock = ch.lock(raf.length()/2+1,raf.length(),SHARED);

   /** Now read the data . . . */

   // release the lock
   sharedLock.release();
  } catch (java.io.IOException ioe) {
   System.err.println(ioe);
  }
  finally {
   if (exclusiveLock != null)
        exclusiveLock.release();
   if (sharedLock != null)
        sharedLock.release();
  }
 }
}
```

Figure 13.2 File-locking example in Java.

all operating systems provide both types of locks: some systems provide only exclusive file locking.

Furthermore, operating systems may provide either **mandatory** or **advisory** file-locking mechanisms. With mandatory locking, once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file. For example, assume a process acquires an exclusive lock on the file `system.log`. If we attempt to open `system.log` from another process—for example, a text editor—the operating system will prevent access until the exclusive lock is released. Alternatively, if the lock is advisory, then the operating system will not prevent the text editor from acquiring access to `system.log`. Rather, the text editor must be written so that it manually acquires the lock before accessing the file. In other words, if the locking scheme is mandatory, the operating system ensures locking integrity. For advisory locking, it is up to software developers to ensure that locks are appropriately acquired and released. As a general rule, Windows operating systems adopt mandatory locking, and UNIX systems employ advisory locks.

The use of file locks requires the same precautions as ordinary process synchronization. For example, programmers developing on systems with mandatory locking must be careful to hold exclusive file locks only while they are accessing the file. Otherwise, they will prevent other processes from accessing the file as well. Furthermore, some measures must be taken to ensure that two or more processes do not become involved in a deadlock while trying to acquire file locks.

13.1.3 File Types

When we design a file system—indeed, an entire operating system—we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to output the binary-object form of a program. This attempt normally produces garbage; however, the attempt can succeed if the operating system has been told that the file is a binary-object program.

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an extension, usually separated by a period (Figure 13.3). In this way, the user and the operating system can tell from the name alone what the type of a file is. Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension made up of additional characters. Examples include `resume.docx`, `server.c`, and `ReaderThread.cpp`.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a `.com`, `.exe`, or `.sh` extension can be executed, for instance. The `.com` and `.exe` files are two forms of binary executable files, whereas the `.sh` file is a **shell script** containing, in ASCII format, commands to the operating system. Application programs also use extensions to indicate file types in which they are interested. For example, Java compilers expect source files to have a `.java` extension, and the Microsoft Word word processor expects its files to end with a `.doc` or `.docx` extension. These extensions are not always required, so a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered "hints" to the applications that operate on them.

Consider, too, the macOS operating system. In this system, each file has a type, such as `.app` (for application). Each file also has a creator attribute

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure 13.3 Common file types.

containing the name of the program that created it. This attribute is set by the operating system during the `create()` call, so its use is enforced and supported by the system. For instance, a file produced by a word processor has the word processor's name as its creator. When the user opens that file, by double-clicking the mouse on the icon representing the file, the word processor is invoked automatically, and the file is loaded, ready to be edited.

The UNIX system uses a **magic number** stored at the beginning of some binary files to indicate the type of data in the file (for example, the format of an image file). Likewise, it uses a text magic number at the start of text files to indicate the type of file (which shell language a script is written in) and so on. (For more details on magic numbers and other computer jargon, see http://www.catb.org/esr/jargon/.) Not all files have magic numbers, so system features cannot be based solely on this information. UNIX does not record the name of the creating program, either. UNIX does allow file-name-extension hints, but these extensions are neither enforced nor depended on by the operating system; they are meant mostly to aid users in determining what type of contents the file contains. Extensions can be used or ignored by a given application, but that is up to the application's programmer.

13.1.4 File Structure

File types also can be used to indicate the internal structure of the file. Source and object files have structures that match the expectations of the programs that read them. Further, certain files must conform to a required structure that is understood by the operating system. For example, the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. Some operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures.

This point brings us to one of the disadvantages of having the operating system support multiple file structures: it makes the operating system large and cumbersome. If the operating system defines five different file structures, it needs to contain the code to support these file structures. In addition, it may be necessary to define every file as one of the file types supported by the operating system. When new applications require information structured in ways not supported by the operating system, severe problems may result.

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

Some operating systems impose (and support) a minimal number of file structures. This approach has been adopted in UNIX, Windows, and others. UNIX considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system. This scheme provides maximum flexibility but little support. Each application program must include its own code to interpret an input file as to the appropriate structure. However, all operating systems must support at least one structure—that of an executable file—so that the system is able to load and run programs.

13.1.5 Internal File Structure

Internally, locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

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

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered a sequence of blocks. All the basic I/O functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem.

Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted. If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted. The waste incurred to keep everything in units of blocks (instead of bytes) is internal fragmentation. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

13.2 Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Others (such as mainframe operating systems) support many access methods, and choosing the right one for a particular application is a major design problem.

13.2.1 Sequential Access

The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

Reads and writes make up the bulk of the operations on a file. A read operation—`read_next()`—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation—`write_next()`—appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning, and on some systems, a program may be able to skip forward or backward n records for some integer n—perhaps only for $n = 1$. Sequential access, which is depicted in Figure 13.4, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

13.2.2 Direct Access

Another method is **direct access** (or **relative access**). Here, a file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus,

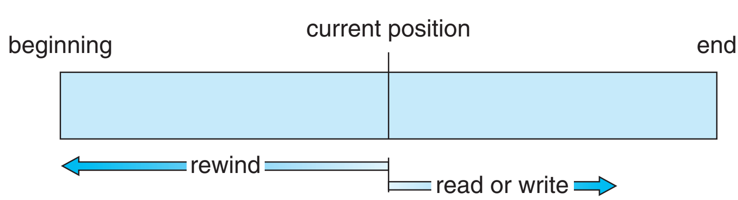


Figure 13.4 Sequential-access file.

we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

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

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

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have `read(n)`, where n is the block number, rather than `read_next()`, and `write(n)` rather than `write_next()`. An alternative approach is to retain `read_next()` and `write_next()` and to add an operation `position_file(n)` where n is the block number. Then, to effect a `read(n)`, we would `position_file(n)` and then `read_next()`.

The block number provided by the user to the operating system is normally a **relative block number**. A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the absolute disk address may be 14703 for the first block and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed (called the **allocation problem**, as we discuss in Chapter 14) and helps to prevent the user from accessing portions of the file system that may not be part of her file. Some systems start their relative block numbers at 0; others start at 1.

How, then, does the system satisfy a request for record N in a file? Assuming we have a logical record length L, the request for record N is turned into an I/O request for L bytes starting at location $L * (N)$ within the file (assuming the first record is $N = 0$). Since logical records are of a fixed size, it is also easy to read, write, or delete a record.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created. Such a file can be accessed only in a manner consistent with its declaration. We can easily simulate sequential access on a direct-access file by simply keeping a variable *cp* that defines our current position, as shown in Figure 13.5. Simulating a direct-access file on a sequential-access file, however, is extremely inefficient and clumsy.

13.2.3 Other Access Methods

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

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp; cp = cp + 1;
write_next	write cp; cp = cp + 1;

Figure 13.5 Simulation of sequential access on a direct-access file.

For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each record consists of a 10-digit UPC and a 6-digit price, for a 16-byte record. If our disk has 1,024 bytes per block, we can store 64 records per block. A file of 120,000 records would occupy about 2,000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block. This index would have 2,000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory. To find the price of a particular item, we can make a binary search of the index. From this search, we learn exactly which block contains the desired record and access that block. This structure allows us to search a large file doing little I/O.

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

For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. Figure 13.6 shows a similar situation as implemented by OpenVMS index and relative files.

13.3 Directory Structure

The directory can be viewed as a symbol table that translates file names into their file control blocks. If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. In this section, we examine several schemes for defining the logical structure of the directory system.

When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

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

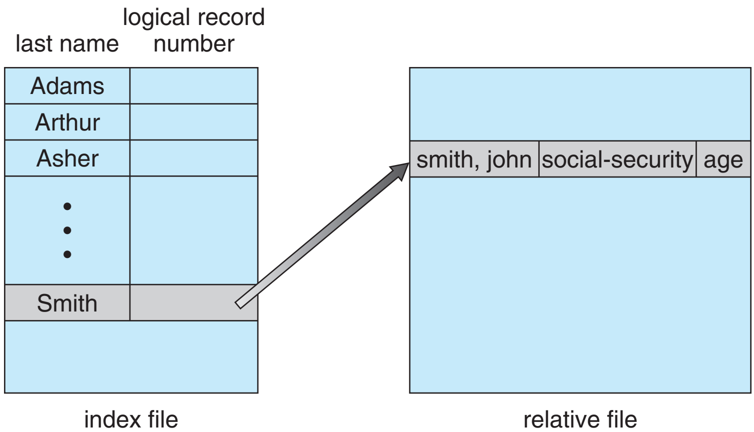


Figure 13.6 Example of index and relative files.

names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.

- **Create a fil** . New files need to be created and added to the directory.
- **Delete a file**. When a file is no longer needed, we want to be able to remove it from the directory. Note a delete leaves a hole in the directory structure and the file system may have a method to defragement the directory structure.
- **List a directory**. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
- **Rename a file**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape, other secondary storage, or across a network to another system or the cloud. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied the backup target and the disk space of that file released for reuse by another file.

In the following sections, we describe the most common schemes for defining the logical structure of a directory.

13.3.1 Single-Level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 13.7).

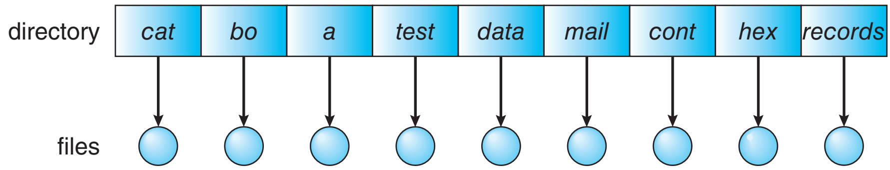


Figure 13.7 Single-level directory.

A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names. If two users call their data file `test.txt`, then the unique-name rule is violated. For example, in one programming class, 23 students called the program for their second assignment `prog2.c`; another 11 called it `assign2.c`. Fortunately, most file systems support file names of up to 255 characters, so it is relatively easy to select unique file names.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

13.3.2 Two-Level Directory

As we have seen, a single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

In the two-level directory structure, each user has his own **user fil directory** (**UFD**). The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's **master fil directory** (**MFD**) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user (Figure 13.8).

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique. To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name

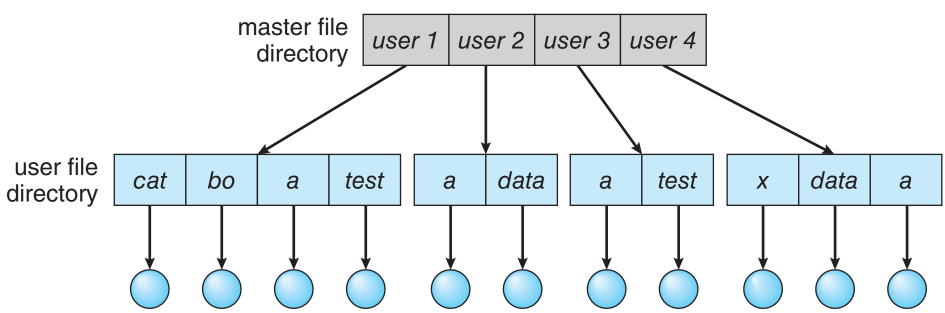


Figure 13.8 Two-level directory structure.

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

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new UFD and adds an entry for it to the MFD. The execution of this program might be restricted to system administrators. The allocation of disk space for user directories can be handled with the techniques discussed in Chapter 14 for files themselves.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

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

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

Additional syntax is needed to specify the volume of a file. For instance, in Windows a volume is specified by a letter followed by a colon. Thus, a file specification might be `C:\userb\test`. Some systems go even further and separate the volume, directory name, and file name parts of the specification. In OpenVMS, for instance, the file `login.com` might be specified as: `u:[sst.crissmeyer]login.com;1`, where `u` is the name of the volume, `sst` is the name of the directory, `crissmeyer` is the name of the subdirectory, and `1` is the version number. Other systems—such as UNIX and Linux—simply treat the volume name as part of the directory name. The first name given is that of the volume, and the rest is the directory and file. For instance, `/u/pgalvin/test` might specify volume `u`, directory `pgalvin`, and file `test`.

A special instance of this situation occurs with the system files. Programs provided as part of the system—loaders, assemblers, compilers, utility routines, libraries, and so on—are generally defined as files. When the appropriate commands are given to the operating system, these files are read by the loader and executed. Many command interpreters simply treat such a command as the name of a file to load and execute. In the directory system as we defined it above, this file name would be searched for in the current UFD. One solution would be to copy the system files into each UFD. However, copying all the system files would waste an enormous amount of space. (If the system files require 5 MB, then supporting 12 users would require 5 × 12 = 60 MB just for copies of the system files.)

The standard solution is to complicate the search procedure slightly. A special user directory is defined to contain the system files (for example, user 0). Whenever a file name is given to be loaded, the operating system first searches the local UFD. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files. The sequence of directories searched when a file is named is called the **search path**. The search path can be extended to contain an unlimited list of directories to search when a command name is given. This method is the one most used in UNIX and Windows. Systems can also be designed so that each user has his own search path.

13.3.3 Tree-Structured Directories

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (Figure 13.9). This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

A directory (or subdirectory) contains a set of files or subdirectories. In many implementations, a directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special

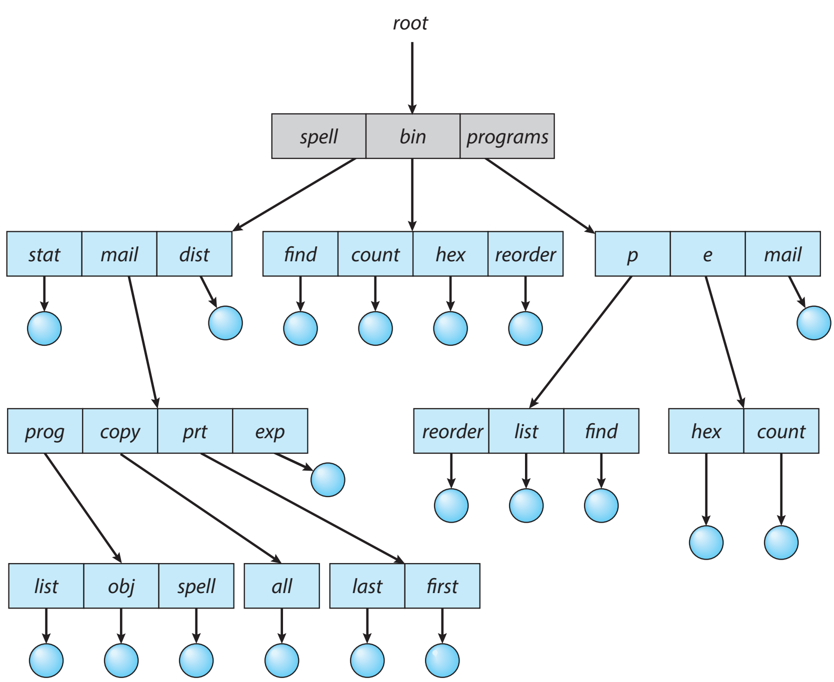


Figure 13.9 Tree-structured directory structure.

system calls are used to create and delete directories. In this case the operating system (or the file system code) implements another file format, that of a directory.

In normal use, each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file. To change directories, a system call could be provided that takes a directory name as a parameter and uses it to redefine the current directory. Thus, the user can change her current directory whenever she wants. Other systems leave it to the application (say, a shell) to track and operate on a current directory, as each process could have different current directories.

The initial current directory of a user's login shell is designated when the user job starts or the user logs in. The operating system searches the accounting file (or some other predefined location) to find an entry for this user (for accounting purposes). In the accounting file is a pointer to (or the name of) the user's initial directory. This pointer is copied to a local variable for this user that specifies the user's initial current directory. From that shell, other processes can be spawned. The current directory of any subprocess is usually the current directory of the parent when it was spawned.

Path names can be of two types: absolute and relative. In UNIX and Linux, an **absolute path name** begins at the root (which is designated by an initial "/") and follows a path down to the specified file, giving the directory names on the path. A **relative path name** defines a path from the current directory. For example, in the tree-structured file system of Figure 13.9, if the current directory is `/spell/mail`, then the relative path name `prt/first` refers to the same file as does the absolute path name `/spell/mail/prt/first`.

Allowing a user to define her own subdirectories permits her to impose a structure on her files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book) or different forms of information. For example, the directory `programs` may contain source programs; the directory `bin` may store all the binaries. (As a side note, executable files were known in many systems as "binaries" which led to them being stored in the `bin` directory.)

An interesting policy decision in a tree-structured directory concerns how to handle the deletion of a directory. If a directory is empty, its entry in the directory that contains it can simply be deleted. However, suppose the directory to be deleted is not empty but contains several files or subdirectories. One of two approaches can be taken. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also. This approach can result in a substantial amount of work. An alternative approach, such as that taken by the UNIX `rm` command, is to provide an option: when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Either approach is fairly easy to implement; the choice is one of policy. The latter policy is more convenient, but it is also more dangerous, because an entire directory structure can be removed with one command. If that command is issued in error, a large number of files and directories will need to be restored (assuming a backup exists).

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users. For example, user B can access a file of user A by specifying its path name. User B can specify either an absolute or a relative path name. Alternatively, user B can change her current directory to be user A's directory and access the file by its file name.

13.3.4 Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. In this situation, the common subdirectory should be ***shared.*** A shared directory or file exists in the file system in two (or more) places at once.

A tree structure prohibits the sharing of files or directories. An **acyclic graph**—that is, a graph with no cycles—allows directories to share subdirectories and files (Figure 13.10). The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is

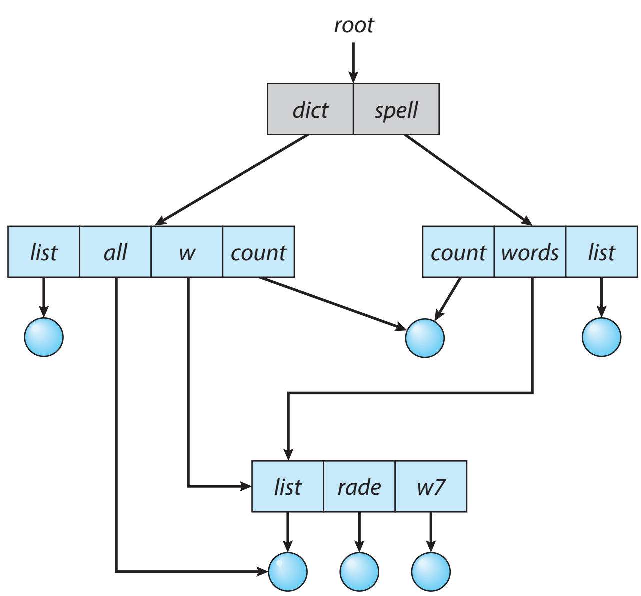


Figure 13.10 Acyclic-graph directory structure.

particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

When people are working as a team, all the files they want to share can be put into one directory. The home directory of each team member could contain this directory of shared files as a subdirectory. Even in the case of a single user, the user's file organization may require that some file be placed in different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project.

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

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

An acyclic-graph directory structure is more flexible than a simple tree structure, but it is also more complex. Several problems must be considered carefully. A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system—to find a file, to accumulate statistics on all files, or to copy all files to backup storage—this problem becomes significant, since we do not want to traverse shared structures more than once.

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

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

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

The trouble with this approach is the variable and potentially large size of the file-reference list. However, we really do not need to keep the entire list —we need to keep only a count of the number of references. Adding a new link or directory entry increments the reference count. Deleting a link or entry decrements the count. When the count is 0, the file can be deleted; there are no remaining references to it. The UNIX operating system uses this approach for nonsymbolic links (or **hard links**), keeping a reference count in the file information block (or inode; see Section C.7.2). By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid problems such as the ones just discussed, some systems simply do not allow shared directories or links.

13.3.5 General Graph Directory

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links, the tree structure is destroyed, resulting in a simple graph structure (Figure 13.11).

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

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to limit arbitrarily the number of directories that will be accessed during a search.

A similar problem exists when we are trying to determine when a file can be deleted. With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing (or a cycle) in the directory structure. In this case, we generally need to use a **garbage collection**

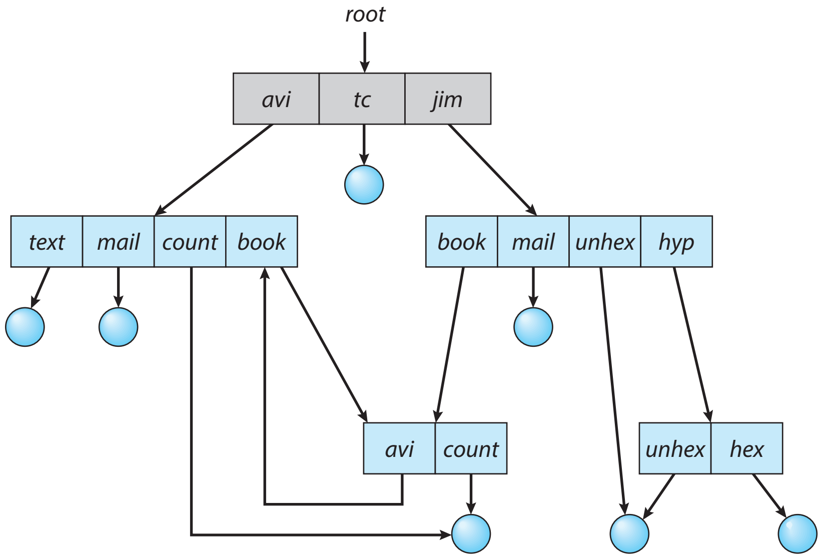


Figure 13.11 General graph directory.

scheme to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. (A similar marking procedure can be used to ensure that a traversal or search will cover everything in the file system once and only once.) Garbage collection for a disk-based file system, however, is extremely time consuming and is thus seldom attempted.

Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles as new links are added to the structure. How do we know when a new link will complete a cycle? There are algorithms to detect cycles in graphs; however, they are computationally expensive, especially when the graph is on disk storage. A simpler algorithm in the special case of directories and links is to bypass links during directory traversal. Cycles are avoided, and no extra overhead is incurred.

13.4 Protection

When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost. Reliability was covered in more detail in Chapter 11.

Protection can be provided in many ways. For a laptop system running a modern operating system, we might provide protection by requiring a user name and password authentication to access it, encrypting the secondary storage so even someone opening the laptop and removing the drive would have a difficult time accessing its data, and firewalling network access so that when it is in use it is difficult to break in via its network connection. In multiuser system, even valid access of the system needs more advanced mechanisms to allow only valid access of the data.

13.4.1 Types of Access

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is controlled access.

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

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

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

Many protection mechanisms have been proposed. Each has advantages and disadvantages and must be appropriate for its intended application. A small computer system that is used by only a few members of a research group, for example, may not need the same types of protection as a large corporate computer that is used for research, finance, and personnel operations. We discuss some approaches to protection in the following sections and present a more complete treatment in Chapter 17.

13.4.2 Access Control

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an **access-control list** (**ACL**) specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

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

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

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

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

The most common recent approach is to combine access-control lists with the more general (and easier to implement) owner, group, and universe access-control scheme just described. For example, Solaris uses the three categories of access by default but allows access-control lists to be added to specific files and directories when more fine-grained access control is desired.

To illustrate, consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named `book.tex.` The protection associated with this file is as follows:

- Sara should be able to invoke all operations on the file.
- Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
- All other users should be able to read, but not write, the file. (Sara is interested in letting as many people as possible read the text so that she can obtain feedback.)

PERMISSIONS IN A UNIX SYSTEM

In the UNIX system, directory protection and file protection are handled similarly. Associated with each file and directory are three fields—owner, group, and universe—each consisting of the three bits `rwx`, where `r` controls **r**ead access, `w` controls **w**rite access, and `x` controls e**x**ecution. Thus, a user can list the content of a subdirectory only if the `r` bit is set in the appropriate field. Similarly, a user can change his current directory to another current directory (say, `foo`) only if the `x` bit associated with the `foo` subdirectory is set in the appropriate field.

A sample directory listing from a UNIX environment is shown in below:

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 jwg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2017	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2017	program
drwx--x--x	4 tag	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

The first field describes the protection of the file or directory. A `d` as the first character indicates a subdirectory. Also shown are the number of links to the file, the owner's name, the group's name, the size of the file in bytes, the date of last modification, and finally the file's name (with optional extension).

To achieve such protection, we must create a new group—say, `text`—with members Jim, Dawn, and Jill. The name of the group, `text`, must then be associated with the file `book.tex`, and the access rights must be set in accordance with the policy we have outlined.

Now consider a visitor to whom Sara would like to grant temporary access to Chapter 1. The visitor cannot be added to the `text` group because that would give him access to all chapters. Because a file can be in only one group, Sara cannot add another group to Chapter 1. With the addition of access-control-list functionality, though, the visitor can be added to the access control list of Chapter 1.

For this scheme to work properly, permissions and access lists must be controlled tightly. This control can be accomplished in several ways. For example, in the UNIX system, groups can be created and modified only by the manager of the facility (or by any superuser). Thus, control is achieved through human interaction. Access lists are discussed further in Section 17.6.2.

With the more limited protection classification, only three fields are needed to define protection. Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it. For example, the UNIX system defines three fields of three bits each—`rwx`, where `r` controls read access, `w` controls write access, and `x` controls execution. A separate field is kept for the file owner, for the file's group, and for all other users. In this scheme, nine bits per file are needed to record protection information. Thus, for our example, the protection fields for the file `book.tex` are as follows: for the owner Sara, all bits are set; for the group `text`, the `r` and `w` bits are set; and for the universe, only the `r` bit is set.

One difficulty in combining approaches comes in the user interface. Users must be able to tell when the optional ACL permissions are set on a file. In the Solaris example, a "+" is appended to the regular permissions, as in:

```
19 -rw-r--r--+ 1 jim staff 130 May 25 22:13 file1
```

A separate set of commands, `setfacl` and `getfacl`, is used to manage the ACLs.

Windows users typically manage access-control lists via the GUI. Figure 13.12 shows a file-permission window on Windows 7 NTFS file system. In this example, user "guest" is specifically denied access to the file `ListPanel.java`.

Another difficulty is assigning precedence when permission and ACLs conflict. For example, if Walter is in a file's group, which has read permission, but the file has an ACL granting Walter read and write permission, should a write by Walter be granted or denied? Solaris and other operating systems give ACLs precedence (as they are more fine-grained and are not assigned by default). This follows the general rule that specificity should have priority.

13.4.3 Other Protection Approaches

Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled in the same way. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages, however. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Second, if only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis. Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to address this problem. More commonly encryption of a partition or individual files provides strong protection, but password management is key.

In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories; that is, we need to provide a mechanism for directory protection. The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory. In addition, we probably want to control whether a user can determine the existence of a file in a directory. Sometimes, knowledge of the existence and name of a file is significant in itself. Thus, listing the contents of a directory must be a protected operation. Similarly, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic and general graphs), a given user may have different access rights to a particular file, depending on the path name used.

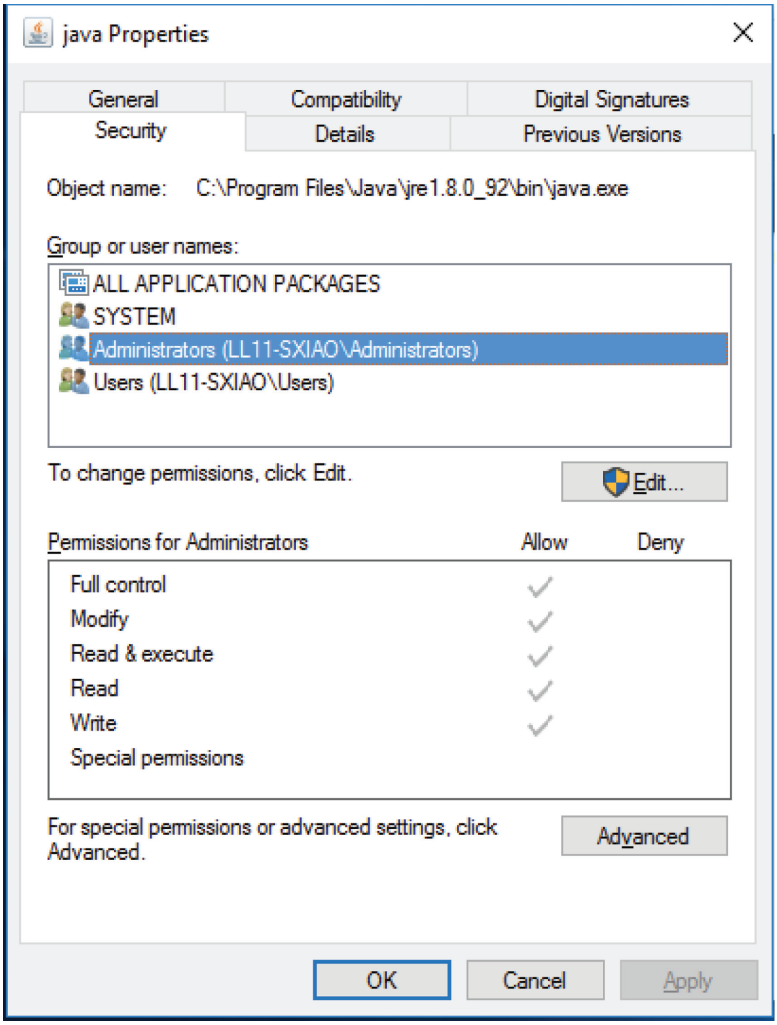


Figure 13.12 Windows 10 access-control list management.

13.5 Memory-Mapped Files

There is one other method of accessing files, and it is very commonly used. Consider a sequential read of a file on disk using the standard system calls `open()`, `read()`, and `write()`. Each file access requires a system call and disk access. Alternatively, we can use the virtual memory techniques discussed in Chapter 10 to treat file I/O as routine memory accesses. This approach, known as **memory mapping** a file, allows a part of the virtual address space to be logically associated with the file. As we shall see, this can lead to significant performance increases.

13.5.1 Basic Mechanism

Memory mapping a file is accomplished by mapping a disk block to a page (or pages) in memory. Initial access to the file proceeds through ordinary demand paging, resulting in a page fault. However, a page-sized portion of the file is read from the file system into a physical page (some systems may opt to read in more than a page-sized chunk of memory at a time). Subsequent reads and writes to the file are handled as routine memory accesses. Manipulating files through memory rather than incurring the overhead of using the `read()` and `write()` system calls simplifies and speeds up file access and usage.

Note that writes to the file mapped in memory are not necessarily immediate (synchronous) writes to the file on secondary storage. Generally, systems update the file based on changes to the memory image only when the file is closed. Under memory pressure, systems will have any intermediate changes to swap space to not lose them when freeing memory for other uses. When the file is closed, all the memory-mapped data are written back to the file on secondary storage and removed from the virtual memory of the process.

Some operating systems provide memory mapping only through a specific system call and use the standard system calls to perform all other file I/O. However, some systems choose to memory-map a file regardless of whether the file was specified as memory-mapped. Let's take Solaris as an example. If a file is specified as memory-mapped (using the `mmap()` system call), Solaris maps the file into the address space of the process. If a file is opened and accessed using ordinary system calls, such as `open()`, `read()`, and `write()`, Solaris still memory-maps the file; however, the file is mapped to the kernel address space. Regardless of how the file is opened, then, Solaris treats all file I/O as memory-mapped, allowing file access to take place via the efficient memory subsystem and avoiding system call overhead caused by each traditional `read()` and `write()`.

Multiple processes may be allowed to map the same file concurrently, to allow sharing of data. Writes by any of the processes modify the data in virtual memory and can be seen by all others that map the same section of the file. Given our earlier discussions of virtual memory, it should be clear how the sharing of memory-mapped sections of memory is implemented: the virtual memory map of each sharing process points to the same page of physical memory—the page that holds a copy of the disk block. This memory sharing is illustrated in Figure 13.13. The memory-mapping system calls can also support copy-on-write functionality, allowing processes to share a file in read-only mode but to have their own copies of any data they modify. So that access to the shared data is coordinated, the processes involved might use one of the mechanisms for achieving mutual exclusion described in Chapter 6.

Quite often, shared memory is in fact implemented by memory mapping files. Under this scenario, processes can communicate using shared memory by having the communicating processes memory-map the same file into their virtual address spaces. The memory-mapped file serves as the region of shared memory between the communicating processes (Figure 13.14). We have already seen this in Section 3.5, where a POSIX shared-memory object is created and each communicating process memory-maps the object into its address space. In the following section, we discuss support in the Windows API for shared memory using memory-mapped files.

13.5.2 Shared Memory in the Windows API

The general outline for creating a region of shared memory using memory-mapped files in the Windows API involves first creating a **fil mapping** for the

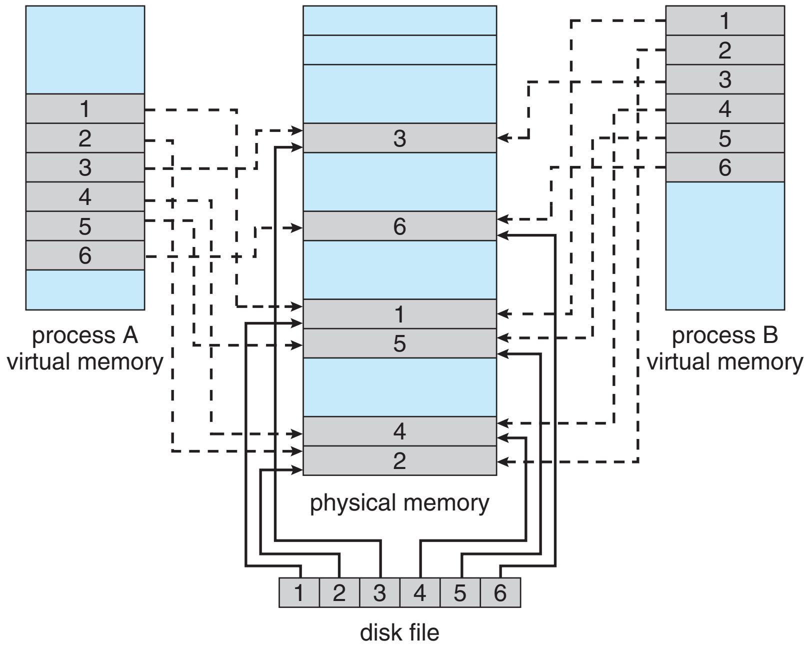


Figure 13.13 Memory-mapped files.

file to be mapped and then establishing a **view** of the mapped file in a process's virtual address space. A second process can then open and create a view of the mapped file in its virtual address space. The mapped file represents the shared-memory object that will enable communication to take place between the processes.

We next illustrate these steps in more detail. In this example, a producer process first creates a shared-memory object using the memory-mapping features available in the Windows API. The producer then writes a message to shared memory. After that, a consumer process opens a mapping to the shared-memory object and reads the message written by the consumer.

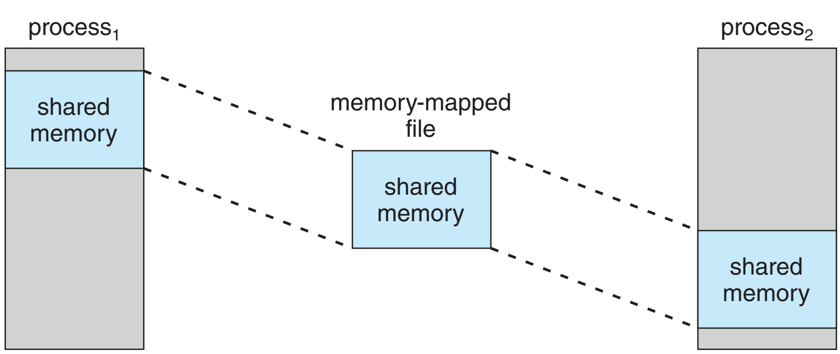


Figure 13.14 Shared memory using memory-mapped I/O.

To establish a memory-mapped file, a process first opens the file to be mapped with the `CreateFile()` function, which returns a `HANDLE` to the opened file. The process then creates a mapping of this file `HANDLE` using the `CreateFileMapping()` function. Once the file mapping is done, the process establishes a view of the mapped file in its virtual address space with the `MapViewOfFile()` function. The view of the mapped file represents the portion of the file being mapped in the virtual address space of the process—the entire file or only a portion of it may be mapped. This sequence in the program

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   HANDLE hFile, hMapFile;
   LPVOID lpMapAddress;

   hFile = CreateFile("temp.txt", /* file name */
      GENERIC_READ | GENERIC_WRITE, /* read/write access */
      0, /* no sharing of the file */
      NULL, /* default security */
      OPEN_ALWAYS, /* open new or existing file */
      FILE_ATTRIBUTE_NORMAL, /* routine file attributes */
      NULL); /* no file template */

   hMapFile = CreateFileMapping(hFile, /* file handle */
      NULL, /* default security */
      PAGE_READWRITE, /* read/write access to mapped pages */
      0, /* map entire file */
      0,
      TEXT("SharedObject")); /* named shared memory object */

   lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
      FILE_MAP_ALL_ACCESS, /* read/write access */
      0, /* mapped view of entire file */
      0,
      0);

   /* write to shared memory */
   sprintf(lpMapAddress,"Shared memory message");

   UnmapViewOfFile(lpMapAddress);
   CloseHandle(hFile);
   CloseHandle(hMapFile);
}
```

Figure 13.15 Producer writing to shared memory using the Windows API.

is shown in Figure 13.15. (We eliminate much of the error checking for code brevity.)

The call to `CreateFileMapping()` creates a **named shared-memory object** called `SharedObject`. The consumer process will communicate using this shared-memory segment by creating a mapping to the same named object. The producer then creates a view of the memory-mapped file in its virtual address space. By passing the last three parameters the value 0, it indicates that the mapped view is the entire file. It could instead have passed values specifying an offset and size, thus creating a view containing only a subsection of the file. (It is important to note that the entire mapping may not be loaded into memory when the mapping is established. Rather, the mapped file may be demand-paged, thus bringing pages into memory only as they are accessed.) The `MapViewOfFile()` function returns a pointer to the shared-memory object; any accesses to this memory location are thus accesses to the memory-mapped file. In this instance, the producer process writes the message "`Shared memory message`" to shared memory.

A program illustrating how the consumer process establishes a view of the named shared-memory object is shown in Figure 13.16. This program is

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   HANDLE hMapFile;
   LPVOID lpMapAddress;

   hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, /* R/W access */
      FALSE, /* no inheritance */
      TEXT("SharedObject")); /* name of mapped file object */

   lpMapAddress = MapViewOfFile(hMapFile, /* mapped object handle */
      FILE_MAP_ALL_ACCESS, /* read/write access */
      0, /* mapped view of entire file */
      0,
      0);

   /* read from shared memory */
   printf("Read message %s", lpMapAddress);

   UnmapViewOfFile(lpMapAddress);
   CloseHandle(hMapFile);
}
```

Figure 13.16 Consumer reading from shared memory using the Windows API.

somewhat simpler than the one shown in Figure 13.15, as all that is necessary is for the process to create a mapping to the existing named shared-memory object. The consumer process must also create a view of the mapped file, just as the producer process did in the program in Figure 13.15. The consumer then reads from shared memory the message "`Shared memory message`" that was written by the producer process.

Finally, both processes remove the view of the mapped file with a call to `UnmapViewOfFile()`. We provide a programming exercise at the end of this chapter using shared memory with memory mapping in the Windows API.

13.6 Summary

- A file is an abstract data type defined and implemented by the operating system. It is a sequence of logical records. A logical record may be a byte, a line (of fixed or variable length), or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.
- A major task for the operating system is to map the logical file concept onto physical storage devices such as hard disk or NVM device. Since the physical record size of the device may not be the same as the logical record size, it may be necessary to order logical records into physical records. Again, this task may be supported by the operating system or left for the application program.
- Within a file system, it is useful to create directories to allow files to be organized. A single-level directory in a multiuser system causes naming problems, since each file must have a unique name. A two-level directory solves this problem by creating a separate directory for each user's files. The directory lists the files by name and includes the file's location on the disk, length, type, owner, time of creation, time of last use, and so on.
- The natural generalization of a two-level directory is a tree-structured directory. A tree-structured directory allows a user to create subdirectories to organize files. Acyclic-graph directory structures enable users to share subdirectories and files but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories but sometimes requires garbage collection to recover unused disk space.
- Remote file systems present challenges in reliability, performance, and security. Distributed information systems maintain user, host, and access information so that clients and servers can share state information to manage use and access.
- Since files are the main information-storage mechanism in most computer systems, file protection is needed on multiuser systems. Access to files can be controlled separately for each type of access—read, write, execute, append, delete, list directory, and so on. File protection can be provided by access lists, passwords, or other techniques.

Practice Exercises

13.1 Some systems automatically delete all user files when a user logs off or a job terminates, unless the user explicitly requests that they be kept. Other systems keep all files unless the user explicitly deletes them. Discuss the relative merits of each approach.

13.2 Why do some systems keep track of the type of a file, while still others leave it to the user and others simply do not implement multiple file types? Which system is “better”?

13.3 Similarly, some systems support many types of structures for a file’s data, while others simply support a stream of bytes. What are the advantages and disadvantages of each approach?

13.4 Could you simulate a multilevel directory structure with a single-level directory structure in which arbitrarily long names can be used? If your answer is yes, explain how you can do so, and contrast this scheme with the multilevel directory scheme. If your answer is no, explain what prevents your simulation’s success. How would your answer change if file names were limited to seven characters?

13.5 Explain the purpose of the `open()` and `close()` operations.

13.6 In some systems, a subdirectory can be read and written by an authorized user, just as ordinary files can be.

a. Describe the protection problems that could arise.

b. Suggest a scheme for dealing with each of these protection problems.

13.7 Consider a system that supports 5,000 users. Suppose that you want to allow 4,990 of these users to be able to access one file.

a. How would you specify this protection scheme in UNIX?

b. Can you suggest another protection scheme that can be used more effectively for this purpose than the scheme provided by UNIX?

13.8 Researchers have suggested that, instead of having an access-control list associated with each file (specifying which users can access the file, and how), we should have a **user control list** associated with each user (specifying which files a user can access, and how). Discuss the relative merits of these two schemes.

Further Reading

A multilevel directory structure was first implemented on the MULTICS system ([Organick (1972)]). Most operating systems now implement multilevel directory structures. These include Linux ([Love (2010)]), macOS ([Singh (2007)]), Solaris ([McDougall and Mauro (2007)]), and all versions of Windows ([Russinovich et al. (2017)]).

A general discussion of Solaris file systems is found in the Sun *System Administration Guide: Devices and File Systems* (http://docs.sun.com/app/docs/doc/817-5093).

The network file system (NFS), designed by Sun Microsystems, allows directory structures to be spread across networked computer systems. NFS Version 4 is described in RFC3505 (http://www.ietf.org/rfc/rfc3530.txt).

A great source of the meanings of computer jargon is http://www.catb.org/esr/jargon/.

Bibliography

[Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

[McDougall and Mauro (2007)] R. McDougall and J. Mauro, *Solaris Internals,* Second Edition, Prentice Hall (2007).

[Organick (1972)] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press (1972).

[Russinovich et al. (2017)] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals–Part 1,* Seventh Edition, Microsoft Press (2017).

[Singh (2007)] A. Singh, *Mac OS X Internals: A Systems Approach*, Addison-Wesley (2007).

Chapter 13 Exercises

13.9 Consider a file system in which a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?

13.10 The open-file table is used to maintain information about files that are currently open. Should the operating system maintain a separate table for each user or maintain just one table that contains references to files that are currently being accessed by all users? If the same file is being accessed by two different programs or users, should there be separate entries in the open-file table? Explain.

13.11 What are the advantages and disadvantages of providing mandatory locks instead of advisory locks whose use is left to users' discretion?

13.12 Provide examples of applications that typically access files according to the following methods:

- Sequential
- Random

13.13 Some systems automatically open a file when it is referenced for the first time and close the file when the job terminates. Discuss the advantages and disadvantages of this scheme compared with the more traditional one, where the user has to open and close the file explicitly.

13.14 If the operating system knew that a certain application was going to access file data in a sequential manner, how could it exploit this information to improve performance?

13.15 Give an example of an application that could benefit from operating-system support for random access to indexed files.

13.16 Some systems provide file sharing by maintaining a single copy of a file. Other systems maintain several copies, one for each of the users sharing the file. Discuss the relative merits of each approach.

CHAPTER 14

File-System Implementation

As we saw in Chapter 13, the file system provides the mechanism for on-line storage and access to file contents, including data and programs. File systems usually reside permanently on secondary storage, which is designed to hold a large amount of data. This chapter is primarily concerned with issues surrounding file storage and access on the most common secondary-storage media, hard disk drives and nonvolatile memory devices. We explore ways to structure file use, to allocate storage space, to recover freed space, to track the locations of data, and to interface other parts of the operating system to secondary storage. Performance issues are considered throughout the chapter.

A given general-purpose operating system provides several file systems. Additionally, many operating systems allow administrators or users to add file systems. Why so many? File systems vary in many respects, including features, performance, reliability, and design goals, and different file systems may serve different purposes. For example, a temporary file system is used for fast storage and retrieval of nonpersistent files, while the default secondary storage file system (such as Linux ext4) sacrifices performance for reliability and features. As we've seen throughout this study of operating systems, there are plenty of choices and variations, making thorough coverage a challenge. In this chapter, we concentrate on the common denominators.

CHAPTER OBJECTIVES

- Describe the details of implementing local file systems and directory structures.
- Discuss block allocation and free-block algorithms and trade-offs.
- Explore file system efficiency and performance issues.
- Look at recovery from file system failures.
- Describe the WAFL file system as a concrete example.

14.1 File-System Structure

Disks provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose:

1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same block.
2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires the drive moving the read–write heads and waiting for the media to rotate.

Nonvolatile memory (NVM) devices are increasingly used for file storage and thus as a location for file systems. They differ from hard disks in that they cannot be rewritten in place and they have different performance characteristics. We discuss disk and NVM-device structure in detail in Chapter 11.

To improve I/O efficiency, I/O transfers between memory and mass storage are performed in units of **blocks**. Each block on a hard disk drive has one or more sectors. Depending on the disk drive, sector size is usually 512 bytes or 4,096 bytes. NVM devices usually have blocks of 4,096 bytes, and the transfer methods used are similar to those used by disk drives.

File systems provide efficient and convenient access to the storage device by allowing data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in Figure 14.1 is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

The **I/O control** level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands, such as "retrieve block 123." Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/O controller's memory to tell the controller which device location to act on and what actions to take. The details of device drivers and the I/O infrastructure are covered in Chapter 12.

The **basic file system** (called the "block I/O subsystem" in Linux) needs only to issue generic commands to the appropriate device driver to read and write blocks on the storage device. It issues commands to the drive based on logical block addresses. It is also concerned with I/O request scheduling. This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks. A block in the buffer is allocated before the transfer of a mass storage block can occur. When the buffer is full, the buffer manager must find more buffer memory or free up buffer space to allow a

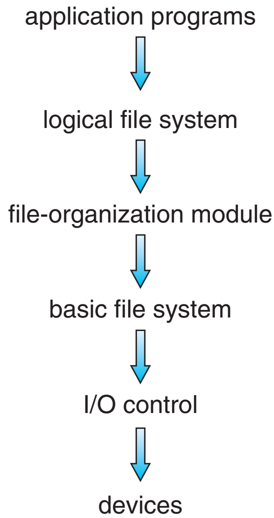


Figure 14.1 Layered file system.

requested I/O to complete. Caches are used to hold frequently used file-system metadata to improve performance, so managing their contents is critical for optimum system performance.

The **file-organizatio module** knows about files and their logical blocks. Each file's logical blocks are numbered from 0 (or 1) through *N*. The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Finally, the **logical file system** manages metadata information. Metadata includes all of the file-system structure except the actual data (or contents of the files). The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file-control blocks. A **file control block** (**FCB**) (an **inode** in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents. The logical file system is also responsible for protection, as discussed in Chapters 13 and 17.

When a layered structure is used for file-system implementation, duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems. Each file system can then have its own logical file-system and file-organization modules. Unfortunately, layering can introduce more operating-system overhead, which may result in decreased performance. The use of layering, including the decision about how many layers to use and what each layer should do, is a major challenge in designing new systems.

Many file systems are in use today, and most operating systems support more than one. For example, most CD-ROMs are written in the ISO 9660 format, a standard format agreed on by CD-ROM manufacturers. In addition to removable-media file systems, each operating system has one or more disk-based file systems. UNIX uses the **UNIX fil system** (**UFS**), which is based on the

Berkeley Fast File System (FFS). Windows supports disk file-system formats of FAT, FAT32, and NTFS (or Windows NT File System), as well as CD-ROM and DVD file-system formats. Although Linux supports over 130 different file systems, the standard Linux file system is known as the **extended file system**, with the most common versions being ext3 and ext4. There are also distributed file systems in which a file system on a server is mounted by one or more client computers across a network.

File-system research continues to be an active area of operating-system design and implementation. Google created its own file system to meet the company's specific storage and retrieval needs, which include high-performance access from many clients across a very large number of disks. Another interesting project is the FUSE file system, which provides flexibility in file-system development and use by implementing and executing file systems as user-level rather than kernel-level code. Using FUSE, a user can add a new file system to a variety of operating systems and can use that file system to manage her files.

14.2 File-System Operations

As was described in Section 13.1.2, operating systems implement `open()` and `close()` systems calls for processes to request access to file contents. In this section, we delve into the structures and operations used to implement file-system operations.

14.2.1 Overview

Several on-storage and in-memory structures are used to implement a file system. These structures vary depending on the operating system and the file system, but some general principles apply.

On storage, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files. Many of these structures are detailed throughout the remainder of this chapter. Here, we describe them briefly:

- A **boot control block** (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume. In UFS, it is called the **boot block**. In NTFS, it is the **partition boot sector**.
- A **volume control block** (per volume) contains volume details, such as the number of blocks in the volume, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers. In UFS, this is called a **superblock**. In NTFS, it is stored in the **master fil table**.
- A directory structure (per file system) is used to organize the files. In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in the master file table.

- A per-file FCB contains many details about the file. It has a unique identifier number to allow association with a directory entry. In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount. Several types of structures may be included.

- An in-memory **mount table** contains information about each mounted volume.
- An in-memory directory-structure cache holds the directory information of recently accessed directories. (For directories at which volumes are mounted, it can contain a pointer to the volume table.)
- The **system-wide open-fil table** contains a copy of the FCB of each open file, as well as other information.
- The **per-process open-fil table** contains pointers to the appropriate entries in the system-wide open-file table, as well as other information, for all files the process has open.
- Buffers hold file-system blocks when they are being read from or written to a file system.

To create a new file, a process calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB. (Alternatively, if the file-system implementation creates all FCBs at file-system creation time, an FCB is allocated from the set of free FCBs.) The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the file system. A typical FCB is shown in Figure 14.2.

Some operating systems, including UNIX, treat a directory exactly the same as a file—one with a "type" field indicating that it is a directory. Other oper-

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Figure 14.2 A typical file-control block.

ating systems, including Windows, implement separate system calls for files and directories and treat directories as entities separate from files. Whatever the larger structural issues, the logical file system can call the file-organization module to map the directory I/O into storage block locations, which are passed on to the basic file system and I/O control system.

14.2.2 Usage

Now that a file has been created, it can be used for I/O. First, though, it must be opened. The `open()` call passes a file name to the logical file system. The `open()` system call first searches the system-wide open-file table to see if the file is already in use by another process. If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table. This algorithm can save substantial overhead. If the file is not already open, the directory structure is searched for the given file name. Parts of the directory structure are usually cached in memory to speed directory operations. Once the file is found, the FCB is copied into a system-wide open-file table in memory. This table not only stores the FCB but also tracks the number of processes that have the file open.

Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields. These other fields may include a pointer to the current location in the file (for the next `read()` or `write()` operation) and the access mode in which the file is open. The `open()` call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer. The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk. It could be cached, though, to save time on subsequent opens of the same file. The name given to the entry varies. UNIX systems refer to it as a **fil descriptor**; Windows refers to it as a **fil handle**.

When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented. When all users that have opened the file close it, any updated metadata are copied back to the disk-based directory structure, and the system-wide open-file table entry is removed.

The caching aspects of file-system structures should not be overlooked. Most systems keep all information about an open file, except for its actual data blocks, in memory. The BSD UNIX system is typical in its use of caches wherever disk I/O can be saved. Its average cache hit rate of 85 percent shows that these techniques are well worth implementing. The BSD UNIX system is described fully in Appendix C.

The operating structures of a file-system implementation are summarized in Figure 14.3.

14.3 Directory Implementation

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system. In this section, we discuss the trade-offs involved in choosing one of these algorithms.

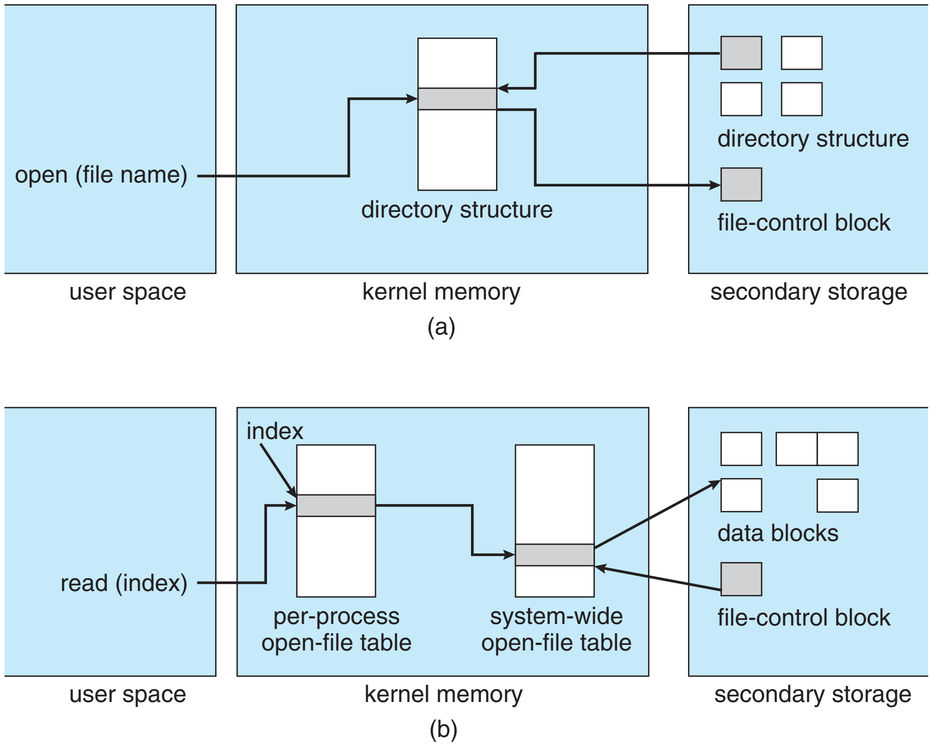


Figure 14.3 In-memory file-system structures. (a) File open. (b) File read.

14.3.1 Linear List

The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time-consuming to execute. To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory. To delete a file, we search the directory for the named file and then release the space allocated to it. To reuse the directory entry, we can do one of several things. We can mark the entry as unused (by assigning it a special name, such as an all-blank name, assigning it an invalid inode number (such as 0), or by including a used–unused bit in each entry), or we can attach it to a list of free directory entries. A third alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory. A linked list can also be used to decrease the time required to delete a file.

The real disadvantage of a linear list of directory entries is that finding a file requires a linear search. Directory information is used frequently, and users will notice if access to it is slow. In fact, many operating systems implement a software cache to store the most recently used directory information. A cache hit avoids the need to constantly reread the information from secondary storage. A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory. A more sophisticated tree data structure, such as a balanced tree, might help here. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

14.3.2 Hash Table

Another data structure used for a file directory is a hash table. Here, a linear list stores the directory entries, but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for collisions—situations in which two file names hash to the same location.

The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size. For example, assume that we make a linear-probing hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63 (for instance, by using the remainder of a division by 64). If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values.

Alternatively, we can use a chained-overflow hash table. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list. Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries. Still, this method is likely to be much faster than a linear search through the entire directory.

14.4 Allocation Methods

The direct-access nature of secondary storage gives us flexibility in the implementation of files. In almost every case, many files are stored on the same device. The main problem is how to allocate space to these files so that storage space is utilized effectively and files can be accessed quickly. Three major methods of allocating secondary storage space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Although some systems support all three, it is more common for a system to use one method for all files within a file-system type.

14.4.1 Contiguous Allocation

Contiguous allocation requires that each file occupy a set of contiguous blocks on the device. Device addresses define a linear ordering on the device. With this ordering, assuming that only one job is accessing the device, accessing block b + 1 after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next. Thus, for HDDs, the number of disk seeks required for accessing contiguously allocated files is

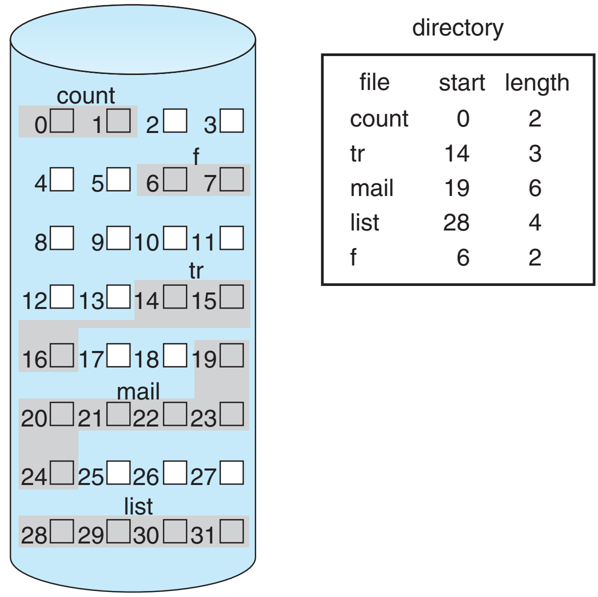


Figure 14.4 Contiguous allocation of disk space.

minimal (assuming blocks with close logical addresses are close physically), as is seek time when a seek is finally needed.

Contiguous allocation of a file is defined by the address of the first block and length (in block units) of the file. If the file is n blocks long and starts at location b, then it occupies blocks b, $b + 1$, $b + 2$, ..., $b + n - 1$. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file (Figure 14.4). Contiguous allocation is easy to implement but has limitations, and is therefore not used in modern file systems.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block $b + i$. Thus, both sequential and direct access can be supported by contiguous allocation.

Contiguous allocation has some problems, however. One difficulty is finding space for a new file. The system chosen to manage free space determines how this task is accomplished; these management systems are discussed in Section 14.5. Any management system can be used, but some are slower than others.

The contiguous-allocation problem can be seen as a particular application of the general **dynamic storage-allocation** problem discussed in Section 9.2, which involves how to satisfy a request of size n from a list of free holes. First fit and best fit are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first fit and best fit are more efficient than worst fit in terms of both time and storage utilization. Neither first fit nor best fit is clearly best in terms of storage utilization, but first fit is generally faster.

All these algorithms suffer from the problem of **external fragmentation**. As files are allocated and deleted, the free storage space is broken into little pieces.

External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

One strategy for preventing loss of significant amounts of storage space to external fragmentation is to copy an entire file system onto another device. The original device is then freed completely, creating one large contiguous free space. We then copy the files back onto the original device by allocating contiguous space from this one large hole. This scheme effectively **compacts** all free space into one contiguous space, solving the fragmentation problem. The cost of this compaction is time, however, and the cost can be particularly high for large storage devices. Compacting these devices may take hours and may be necessary on a weekly basis. Some systems require that this function be done **off-line**, with the file system unmounted. During this **down time**, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines. Most modern systems that need defragmentation can perform it **on-line** during normal system operations, but the performance penalty can be substantial.

Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. How does the creator (program or person) know the size of the file to be created? In some cases, this determination may be fairly simple (copying an existing file, for example). In general, however, the size of an output file may be difficult to estimate.

If we allocate too little space to a file, we may find that the file cannot be extended. Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place. Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly. To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space. The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions can be repeated as long as space exists, although it can be time consuming. The user need never be informed explicitly about what is happening, however; the system continues despite the problem, although more and more slowly.

Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient. A file that will grow slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space will be unused for a long time. The file therefore has a large amount of internal fragmentation.

To minimize these drawbacks, an operating system can use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially. Then, if that amount proves not to be large enough, another chunk of contiguous space, known as an **extent**, is added. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent. On some systems, the owner of the file can set the extent size, but this setting results in inefficiencies if the owner is incorrect. Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can become a problem as extents of varying sizes are allocated and deallocated. The commercial Symantec Veritas file system uses extents to optimize performance. Veritas is a high-performance replacement for the standard UNIX UFS.

14.4.2 Linked Allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of storage blocks; the blocks may be scattered anywhere on the device. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 (Figure 14.5). Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a block address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first block of the file. This pointer is initialized to `null` (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the *i*th

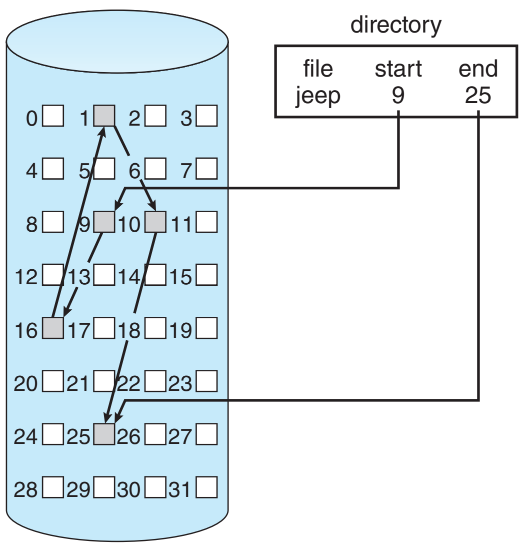


Figure 14.5 Linked allocation of disk space.

block of a file, we must start at the beginning of that file and follow the pointers until we get to the *i*th block. Each access to a pointer requires a storage device read, and some require an HDD seek. Consequently, it is inefficient to support a direct-access capability for linked-allocation files.

Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space than it would otherwise.

The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate clusters rather than blocks. For instance, the file system may define a cluster as four blocks and operate on the secondary storage device only in cluster units. Pointers then use a much smaller percentage of the file's space. This method allows the logical-to-physical block mapping to remain simple but improves HDD throughput (because fewer disk-head seeks are required) and decreases the space needed for block allocation and free-list management. The cost of this approach is an increase in internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full. Also random I/O performance suffers because a request for a small amount of data transfers a large amount of data. Clusters can be used to improve the disk-access time for many other algorithms as well, so they are used in most file systems.

Yet another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the device, and consider what would happen if a pointer was lost or damaged. A bug in the operating-system software or a hardware failure might result in picking up the wrong pointer. This error could in turn result in linking into the free-space list or into another file. One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block. However, these schemes require even more overhead for each file.

An important variation on linked allocation is the use of a **file-allocatio table** (**FAT**). This simple but efficient method of disk-space allocation was used by the MS-DOS operating system. A section of storage at the beginning of each volume is set aside to contain the table. The table has one entry for each block and is indexed by block number. The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block, which has a special end-of-file value as the table entry. An unused block is indicated by a table value of 0. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure shown in Figure 14.6 for a file consisting of disk blocks 217, 618, and 339.

The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached. The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself. In the worst case, both moves occur for each of the blocks. A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

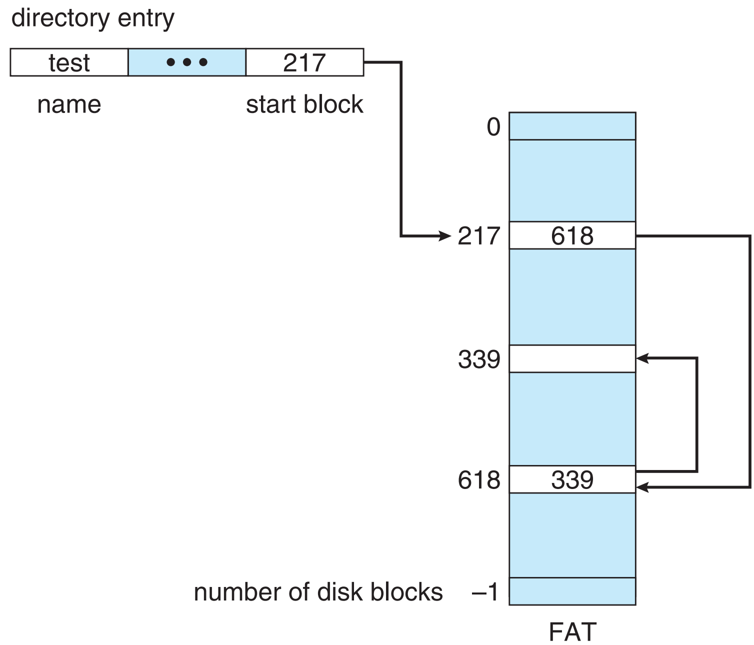


Figure 14.6 File-allocation table.

14.4.3 Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block**.

Each file has its own index block, which is an array of storage-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block (Figure 14.7). To find and read the ith block, we use the pointer in the ith index-block entry. This scheme is similar to the paging scheme described in Section 9.3.

When the file is created, all pointers in the index block are set to `null`. When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the ith index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the storage device can satisfy a request for more space. Indexed allocation does suffer from wasted space, however. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block. With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-`null`.

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as

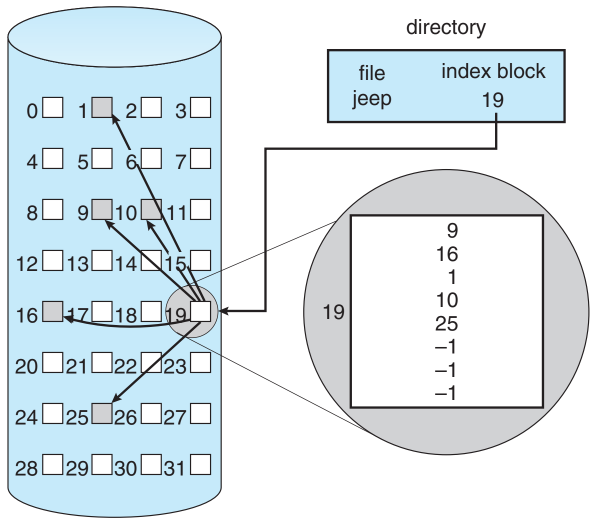


Figure 14.7 Indexed allocation of disk space.

possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include the following:

- **Linked scheme**. An index block is normally one storage block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is `null` (for a small file) or is a pointer to another index block (for a large file).

- **Multilevel index**. A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size. With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.

- **Combined scheme**. Another alternative, used in UNIX-based file systems, is to keep the first, say, 15 pointers of the index block in the file's inode. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The next three pointers point to **indirect blocks**. The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a **triple indirect block**. (A UNIX inode is shown in Figure 14.8.)

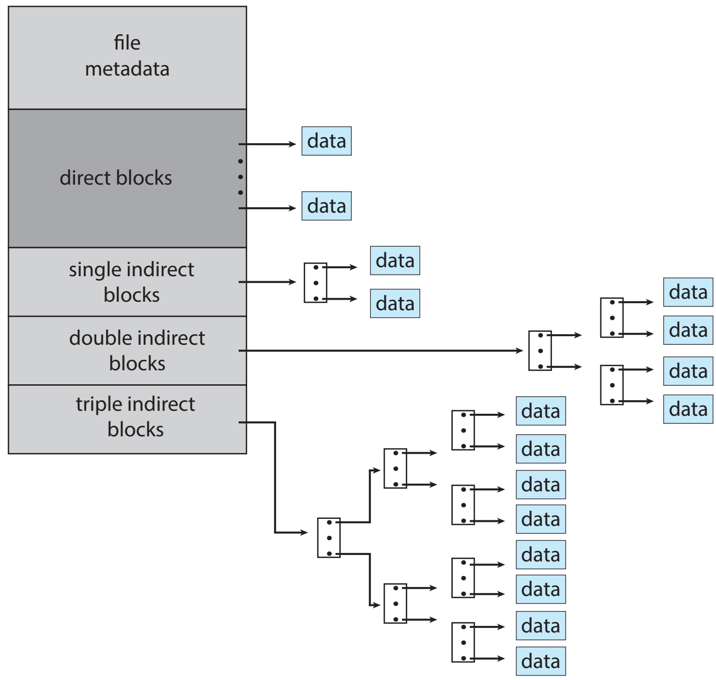


Figure 14.8 The UNIX inode.

Under this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the 4-byte file pointers used by many operating systems. A 32-bit file pointer reaches only 2^{32} bytes, or 4 GB. Many UNIX and Linux implementations now support 64-bit file pointers, which allows files and file systems to be several exbibytes in size. The ZFS file system supports 128-bit file pointers.

Indexed-allocation schemes suffer from some of the same performance problems as does linked allocation. Specifically, the index blocks can be cached in memory, but the data blocks may be spread all over a volume.

14.4.4 Performance

The allocation methods that we have discussed vary in their storage efficiency and data-block access times. Both are important criteria in selecting the proper method or methods for an operating system to implement.

Before selecting an allocation method, we need to determine how the systems will be used. A system with mostly sequential access should not use the same method as a system with mostly random access.

For any type of access, contiguous allocation requires only one access to get a block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the address of the ith block (or the next block) and read it directly.

For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access; for direct access, however, an access to the ith block might require i block reads. This problem indicates why linked allocation should not be used for an application requiring direct access.

As a result, some systems support direct-access files by using contiguous allocation and sequential-access files by using linked allocation. For these systems, the type of access to be made must be declared when the file is created. A file created for sequential access will be linked and cannot be used for direct access. A file created for direct access will be contiguous and can support both direct access and sequential access, but its maximum length must be declared when it is created. In this case, the operating system must have appropriate data structures and algorithms to support both allocation methods. Files can be converted from one type to another by the creation of a new file of the desired type, into which the contents of the old file are copied. The old file may then be deleted and the new file renamed.

Indexed allocation is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space. If this memory space is not available, then we may have to read first the index block and then the desired data block. For a two-level index, two index-block reads might be necessary. For an extremely large file, accessing a block near the end of the file would require reading in all the index blocks before the needed data block finally could be read. Thus, the performance of indexed allocation depends on the index structure, on the size of the file, and on the position of the block desired.

Some systems combine contiguous allocation with indexed allocation by using contiguous allocation for small files (up to three or four blocks) and automatically switching to an indexed allocation if the file grows large. Since most files are small, and contiguous allocation is efficient for small files, average performance can be quite good.

Many other optimizations are in use. Given the disparity between CPU speed and disk speed, it is not unreasonable to add thousands of extra instructions to the operating system to save just a few disk-head movements. Furthermore, this disparity is increasing over time, to the point where hundreds of thousands of instructions could reasonably be used to optimize head movements.

For NVM devices, there are no disk head seeks, so different algorithms and optimizations are needed. Using an old algorithm that spends many CPU cycles trying to avoid a nonexistent head movement would be very inefficient. Existing file systems are being modified and new ones being created to attain maximum performance from NVM storage devices. These developments aim to reduce the instruction count and overall path between the storage device and application access to the data.

14.5 Free-Space Management

Since storage space is limited, we need to reuse the space from deleted files for new files, if possible. (Write-once optical disks allow only one write to any given sector, and thus reuse is not physically possible.) To keep track of free disk space, the system maintains a **free-space list**. The free-space list records all free device blocks—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its space is added to the free-space list. The free-space list, despite its name, is not necessarily implemented as a list, as we discuss next.

14.5.1 Bit Vector

Frequently, the free-space list is implemented as a **bitmap** or **bit vector**. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be

001111001111110001100000011100000 ...

The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk. Indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose. One technique for finding the first free block on a system that uses a bit vector to allocate space is to sequentially check each word in the bitmap to see whether that value is not 0, since a 0-valued word contains only 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is

(number of bits per word) × (number of 0-value words) + offset of first 1 bit.

Again, we see hardware features driving software functionality. Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory (and is written to the device containing the file system occasionally for recovery needs). Keeping it in main memory is possible for smaller devices but not necessarily for larger ones. A 1.3-GB disk with 512-byte blocks would need a bitmap of over 332 KB to track its free blocks, although clustering the blocks in groups of four reduces this number to around 83 KB per disk. A 1-TB disk with 4-KB blocks would require 32 MB (2^{40} / 2^{12} = 2^{28} bits = 2^{25} bytes = 2^{5} MB) to store its bitmap. Given that disk size constantly increases, the problem with bit vectors will continue to escalate as well.

14.5.2 Linked List

Another approach to free-space management is to link together all the free blocks, keeping a pointer to the first free block in a special location in the file system and caching it in memory. This first block contains a pointer to the next free block, and so on. Recall our earlier example (Section 14.5.1), in which blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated. In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on (Figure 14.9). This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time on HDDs. Fortunately, however, traversing the free list is not a frequent action. Usually,

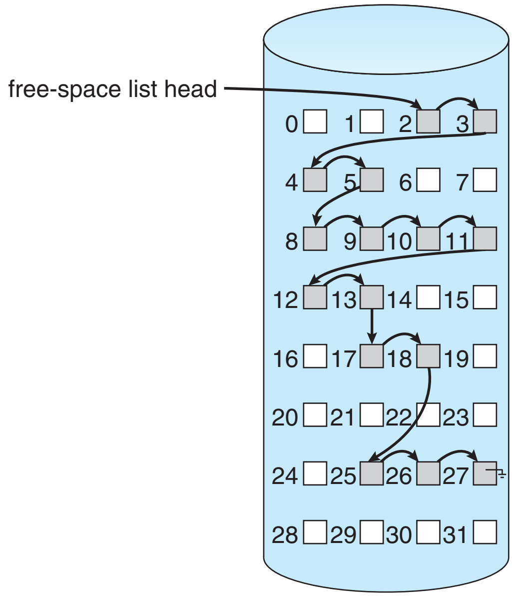


Figure 14.9 Linked free-space list on disk.

the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free-block accounting into the allocation data structure. No separate method is needed.

14.5.3 Grouping

A modification of the free-list approach stores the addresses of n free blocks in the first free block. The first $n-1$ of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on. The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

14.5.4 Counting

Another approach takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of n free block addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a device address and a count. Although each entry requires more space than would a simple disk address, the overall list is shorter, as long as the count is generally greater than 1. Note that this method of tracking free space is similar to the extent method of allocating blocks. These entries can be stored in a balanced tree, rather than a linked list, for efficient lookup, insertion, and deletion.

14.5.5 Space Maps

Oracle's **ZFS** file system (found in Solaris and some other operating systems) was designed to encompass huge numbers of files, directories, and even file systems (in ZFS, we can create file-system hierarchies). On these scales, metadata I/O can have a large performance impact. Consider, for example, that if the free-space list is implemented as a bitmap, bitmaps must be modified both when blocks are allocated and when they are freed. Freeing 1 GB of data on a 1-TB disk could cause thousands of blocks of bitmaps to be updated, because those data blocks could be scattered over the entire disk. Clearly, the data structures for such a system could be large and inefficient.

In its management of free space, ZFS uses a combination of techniques to control the size of data structures and minimize the I/O needed to manage those structures. First, ZFS creates **metaslabs** to divide the space on the device into chunks of manageable size. A given volume may contain hundreds of metaslabs. Each metaslab has an associated space map. ZFS uses the counting algorithm to store information about free blocks. Rather than write counting structures to disk, it uses log-structured file-system techniques to record them. The space map is a log of all block activity (allocating and freeing), in time order, in counting format. When ZFS decides to allocate or free space from a metaslab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset, and replays the log into that structure. The in-memory space map is then an accurate representation of the allocated and free space in the metaslab. ZFS also condenses the map as much as possible by combining contiguous free blocks into a single entry. Finally, the free-space list is updated on disk as part of the transaction-oriented operations of ZFS. During the collection and sorting phase, block requests can still occur, and ZFS satisfies these requests from the log. In essence, the log plus the balanced tree ***is*** the free list.

14.5.6 TRIMing Unused Blocks

HDDs and other storage media that allow blocks to be overwritten for updates need only the free list for managing free space. Blocks do not need to be treated specially when freed. A freed block typically keeps its data (but without any file pointers to the block) until the data are overwritten when the block is next allocated.

Storage devices that do not allow overwrite, such as NVM flash-based storage devices, suffer badly when these same algorithms are applied. Recall from Section 11.1.2 that such devices must be erased before they can again be written to, and that those erases must be made in large chunks (blocks, composed of pages) and take a relatively long time compared with reads or writes.

A new mechanism is needed to allow the file system to inform the storage device that a page is free and can be considered for erasure (once the block containing the page is entirely free). That mechanism varies based on the storage controller. For ATA-attached drives, it is TRIM, while for NVMe-based storage, it is the `unallocate` command. Whatever the specific controller command, this mechanism keeps storage space available for writing. Without such a capability, the storage device gets full and needs garbage collection and block erasure, leading to decreases in storage I/O write performance (known as "a write cliff").

With the TRIM mechanism and similar capabilities, the garbage collection and erase steps can occur before the device is nearly full, allowing the device to provide more consistent performance.

14.6 Efficiency and Performance

Now that we have discussed various block-allocation and directory-management options, we can further consider their effect on performance and efficient storage use. Disks tend to represent a major bottleneck in system performance, since they are the slowest main computer component. Even NVM devices are slow compared with CPU and main memory, so their performance must be optimized as well. In this section, we discuss a variety of techniques used to improve the efficiency and performance of secondary storage.

14.6.1 Efficiency

The efficient use of storage device space depends heavily on the allocation and directory algorithms in use. For instance, UNIX inodes are preallocated on a volume. Even an empty disk has a percentage of its space lost to inodes. However, by preallocating the inodes and spreading them across the volume, we improve the file system's performance. This improved performance results from the UNIX allocation and free-space algorithms, which try to keep a file's data blocks near that file's inode block to reduce seek time.

As another example, let's reconsider the clustering scheme discussed in Section 14.4, which improves file-seek and file-transfer performance at the cost of internal fragmentation. To reduce this fragmentation, BSD UNIX varies the cluster size as a file grows. Large clusters are used where they can be filled, and small clusters are used for small files and the last cluster of a file. This system is described in Appendix C.

The types of data normally kept in a file's directory (or inode) entry also require consideration. Commonly, a "last write date" is recorded to supply information to the user and to determine whether the file needs to be backed up. Some systems also keep a "last access date," so that a user can determine when the file was last read. The result of keeping this information is that, whenever the file is read, a field in the directory structure must be written to. That means the block must be read into memory, a section changed, and the block written back out to the device, because operations on secondary storage occur only in block (or cluster) chunks. So any time a file is opened for reading, its FCB must be read and written as well. This requirement can be inefficient for frequently accessed files, so we must weigh its benefit against its performance cost when designing a file system. Generally, every data item associated with a file needs to be considered for its effect on efficiency and performance.

Consider, for instance, how efficiency is affected by the size of the pointers used to access data. Most systems use either 32-bit or 64-bit pointers throughout the operating system. Using 32-bit pointers limits the size of a file to 2^{32}, or 4 GB. Using 64-bit pointers allows very large file sizes, but 64-bit pointers require more space to store. As a result, the allocation and free-space-management methods (linked lists, indexes, and so on) use more storage space.

One of the difficulties in choosing a pointer size—or, indeed, any fixed allocation size within an operating system—is planning for the effects of changing technology. Consider that the IBM PC XT had a 10-MB hard drive and an MS-DOS FAT file system that could support only 32 MB. (Each FAT entry was 12 bits, pointing to an 8-KB cluster.) As disk capacities increased, larger disks had to be split into 32-MB partitions, because the file system could not track blocks beyond 32 MB. As hard disks with capacities of over 100 MB became common, the disk data structures and algorithms in MS-DOS had to be modified to allow larger file systems. (Each FAT entry was expanded to 16 bits and later to 32 bits.) The initial file-system decisions were made for efficiency reasons; however, with the advent of MS-DOS Version 4, millions of computer users were inconvenienced when they had to switch to the new, larger file system. Solaris's ZFS file system uses 128-bit pointers, which theoretically should never need to be extended. (The minimum mass of a device capable of storing 2^{128} bytes using atomic-level storage would be about 272 trillion kilograms.)

As another example, consider the evolution of the Solaris operating system. Originally, many data structures were of fixed length, allocated at system startup. These structures included the process table and the open-file table. When the process table became full, no more processes could be created. When the file table became full, no more files could be opened. The system would fail to provide services to users. Table sizes could be increased only by recompiling the kernel and rebooting the system. With later releases of Solaris, (as with modern Linux kernels) almost all kernel structures were allocated dynamically, eliminating these artificial limits on system performance. Of course, the algorithms that manipulate these tables are more complicated, and the operating system is a little slower because it must dynamically allocate and deallocate table entries; but that price is the usual one for more general functionality.

14.6.2 Performance

Even after the basic file-system algorithms have been selected, we can still improve performance in several ways. As was discussed in Chapter 12, storage device controllers include local memory to form an on-board cache that is large enough to store entire tracks or blocks at a time. On an HDD, once a seek is performed, the track is read into the disk cache starting at the sector under the disk head (reducing latency time). The disk controller then transfers any sector requests to the operating system. Once blocks make it from the disk controller into main memory, the operating system may cache the blocks there.

Some systems maintain a separate section of main memory for a **buffer cache**, where blocks are kept under the assumption that they will be used again shortly. Other systems cache file data using a **page cache**. The page cache uses virtual memory techniques to cache file data as pages rather than as file-system-oriented blocks. Caching file data using virtual addresses is far more efficient than caching through physical disk blocks, as accesses interface with virtual memory rather than the file system. Several systems—including Solaris, Linux, and Windows—use page caching to cache both process pages and file data. This is known as **unifie virtual memory**.

Some versions of UNIX and Linux provide a **unifie buffer cache**. To illustrate the benefits of the unified buffer cache, consider the two alternatives

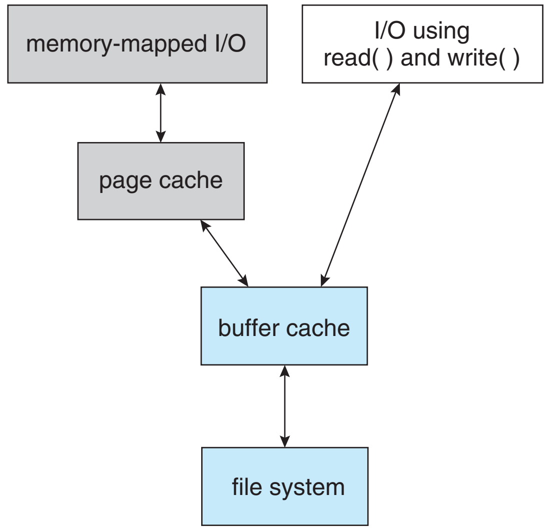


Figure 14.10 I/O without a unified buffer cache.

for opening and accessing a file. One approach is to use memory mapping (Section 13.5); the second is to use the standard system calls `read()` and `write()`. Without a unified buffer cache, we have a situation similar to Figure 14.10. Here, the `read()` and `write()` system calls go through the buffer cache. The memory-mapping call, however, requires using two caches—the page cache and the buffer cache. A memory mapping proceeds by reading in disk blocks from the file system and storing them in the buffer cache. Because the virtual memory system does not interface with the buffer cache, the contents of the file in the buffer cache must be copied into the page cache. This situation, known as **double caching**, requires caching file-system data twice. Not only does this waste memory but it also wastes significant CPU and I/O cycles due to the extra data movement within system memory. In addition, inconsistencies between the two caches can result in corrupt files. In contrast, when a unified buffer cache is provided, both memory mapping and the `read()` and `write()` system calls use the same page cache. This has the benefit of avoiding double caching, and it allows the virtual memory system to manage file-system data. The unified buffer cache is shown in Figure 14.11.

Regardless of whether we are caching storage blocks or pages (or both), least recently used (LRU) (Section 10.4.4) seems a reasonable general-purpose algorithm for block or page replacement. However, the evolution of the Solaris page-caching algorithms reveals the difficulty in choosing an algorithm. Solaris allows processes and the page cache to share unused memory. Versions earlier than Solaris 2.5.1 made no distinction between allocating pages to a process and allocating them to the page cache. As a result, a system performing many I/O operations used most of the available memory for caching pages. Because of the high rates of I/O, the page scanner (Section 10.10.3) reclaimed pages from processes—rather than from the page cache—when free memory ran low. Solaris 2.6 and Solaris 7 optionally implemented priority paging, in which the page scanner gave priority to process pages over the page cache. Solaris 8 applied a fixed limit to process pages and the file-system page cache, prevent-

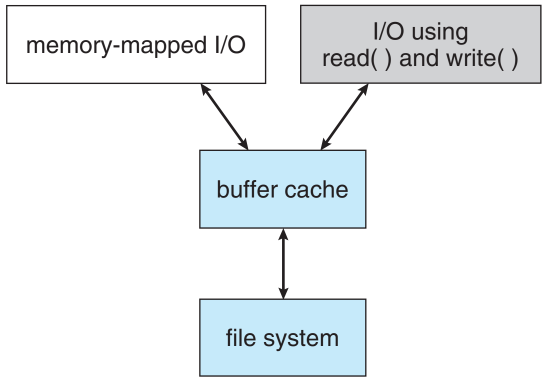


Figure 14.11 I/O using a unified buffer cache.

ing either from forcing the other out of memory. Solaris 9 and 10 again changed the algorithms to maximize memory use and minimize thrashing.

Another issue that can affect the performance of I/O is whether writes to the file system occur synchronously or asynchronously. **Synchronous writes** occur in the order in which the storage subsystem receives them, and the writes are not buffered. Thus, the calling routine must wait for the data to reach the drive before it can proceed. In an **asynchronous write**, the data are stored in the cache, and control returns to the caller. Most writes are asynchronous. However, metadata writes, among others, can be synchronous. Operating systems frequently include a flag in the `open` system call to allow a process to request that writes be performed synchronously. For example, databases use this feature for atomic transactions, to assure that data reach stable storage in the required order.

Some systems optimize their page cache by using different replacement algorithms, depending on the access type of the file. A file being read or written sequentially should not have its pages replaced in LRU order, because the most recently used page will be used last, or perhaps never again. Instead, sequential access can be optimized by techniques known as free-behind and read-ahead. **Free-behind** removes a page from the buffer as soon as the next page is requested. The previous pages are not likely to be used again and waste buffer space. With **read-ahead**, a requested page and several subsequent pages are read and cached. These pages are likely to be requested after the current page is processed. Retrieving these data from the disk in one transfer and caching them saves a considerable amount of time. One might think that a track cache on the controller would eliminate the need for read-ahead on a multiprogrammed system. However, because of the high latency and overhead involved in making many small transfers from the track cache to main memory, performing a read-ahead remains beneficial.

The page cache, the file system, and the device drivers have some interesting interactions. When small amounts of data are written to a file, the pages are buffered in the cache, and the storage device driver sorts its output queue according to device address. These two actions allow a disk driver to minimize disk-head seeks. Unless synchronous writes are required, a process writing to disk simply writes into the cache, and the system asynchronously writes the data to disk when convenient. The user process sees very fast writes. When data are read from a disk file, the block I/O system does some read-ahead; however, writes are much more nearly asynchronous than are reads. Thus, output to the disk through the file system is often faster than is input for small transfers, counter to intuition. No matter how much buffering and caching is available, large, continuous I/O can overrun the capacity and end up bottlenecked on the device's performance. Consider writing a large movie file to a HDD. If the file is larger than the page cache (or the part of the page cache available to the process) then the page cache will fill and all I/O will occur at drive speed. Current HDDs read faster than they write, so in this instance the performance aspects are reversed from smaller I/O performance.

14.7 Recovery

Files and directories are kept both in main memory and on the storage volume, and care must be taken to ensure that a system failure does not result in loss of data or in data inconsistency. A system crash can cause inconsistencies among on-storage file-system data structures, such as directory structures, free-block pointers, and free FCB pointers. Many file systems apply changes to these structures in place. A typical operation, such as creating a file, can involve many structural changes within the file system on the disk. Directory structures are modified, FCBs are allocated, data blocks are allocated, and the free counts for all of these blocks are decreased. These changes can be interrupted by a crash, and inconsistencies among the structures can result. For example, the free FCB count might indicate that an FCB had been allocated, but the directory structure might not point to the FCB. Compounding this problem is the caching that operating systems do to optimize I/O performance. Some changes may go directly to storage, while others may be cached. If the cached changes do not reach the storage device before a crash occurs, more corruption is possible.

In addition to crashes, bugs in file-system implementation, device controllers, and even user applications can corrupt a file system. File systems have varying methods to deal with corruption, depending on the file-system data structures and algorithms. We deal with these issues next.

14.7.1 Consistency Checking

Whatever the cause of corruption, a file system must first detect the problems and then correct them. For detection, a scan of all the metadata on each file system can confirm or deny the consistency of the system. Unfortunately, this scan can take minutes or hours and should occur every time the system boots. Alternatively, a file system can record its state within the file-system metadata. At the start of any metadata change, a status bit is set to indicate that the metadata is in flux. If all updates to the metadata complete successfully, the file system can clear that bit. If, however, the status bit remains set, a consistency checker is run.

The **consistency checker**—a systems program such as `fsck` in UNIX—compares the data in the directory structure and other metadata with the state on storage and tries to fix any inconsistencies it finds. The allocation and free-space-management algorithms dictate what types of problems the checker can find and how successful it will be in fixing them. For instance, if linked allocation is used and there is a link from any block to its next block, then the entire file can be reconstructed from the data blocks, and the directory structure can be recreated. In contrast, the loss of a directory entry on an indexed allocation system can be disastrous, because the data blocks have no knowledge of one another. For this reason, some UNIX file systems cache directory entries for reads, but any write that results in space allocation, or other metadata changes, is done synchronously, before the corresponding data blocks are written. Of course, problems can still occur if a synchronous write is interrupted by a crash. Some NVM storage devices contain a battery or supercapacitor to provide enough power, even during a power loss, to write data from device buffers to the storage media so the data are not lost. But even those precautions do not protect against corruption due to a crash.

14.7.2 Log-Structured File Systems

Computer scientists often find that algorithms and technologies originally used in one area are equally useful in other areas. Such is the case with the database log-based recovery algorithms. These logging algorithms have been applied successfully to the problem of consistency checking. The resulting implementations are known as **log-based transaction-oriented** (or **journaling**) file systems.

Note that with the consistency-checking approach discussed in the preceding section, we essentially allow structures to break and repair them on recovery. However, there are several problems with this approach. One is that the inconsistency may be irreparable. The consistency check may not be able to recover the structures, resulting in loss of files and even entire directories. Consistency checking can require human intervention to resolve conflicts, and that is inconvenient if no human is available. The system can remain unavailable until the human tells it how to proceed. Consistency checking also takes system and clock time. To check terabytes of data, hours of clock time may be required.

The solution to this problem is to apply log-based recovery techniques to file-system metadata updates. Both NTFS and the Veritas file system use this method, and it is included in recent versions of UFS on Solaris. In fact, it is now common on many file systems including ext3, ext4, and ZFS.

Fundamentally, all metadata changes are written sequentially to a log. Each set of operations for performing a specific task is a **transaction**. Once the changes are written to this log, they are considered to be committed, and the system call can return to the user process, allowing it to continue execution. Meanwhile, these log entries are replayed across the actual file-system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, and entry is made in the log indicating that. The log file is is actually a circular buffer. A **circular buffer** writes to the end of its space and then continues at the beginning, overwriting older values as it goes. We would not want the buffer to write over data that had not yet been saved, so that scenario is avoided. The log may be in a separate section of the file system or even on a separate storage device.

If the system crashes, the log file will contain zero or more transactions. Any transactions it contains were not completed to the file system, even though they were committed by the operating system, so they must now be completed. The transactions can be executed from the pointer until the work is complete so that the file-system structures remain consistent. The only problem occurs when a transaction was aborted—that is, was not committed before the system crashed. Any changes from such a transaction that were applied to the file system must be undone, again preserving the consistency of the file system. This recovery is all that is needed after a crash, eliminating any problems with consistency checking.

A side benefit of using logging on disk metadata updates is that those updates proceed much faster than when they are applied directly to the on-disk data structures. The reason is found in the performance advantage of sequential I/O over random I/O. The costly synchronous random metadata writes are turned into much less costly synchronous sequential writes to the log-structured file system's logging area. Those changes, in turn, are replayed asynchronously via random writes to the appropriate structures. The overall result is a significant gain in performance of metadata-oriented operations, such as file creation and deletion, on HDD storage.

14.7.3 Other Solutions

Another alternative to consistency checking is employed by Network Appliance's WAFL file system and the Solaris ZFS file system. These systems never overwrite blocks with new data. Rather, a transaction writes all data and metadata changes to new blocks. When the transaction is complete, the metadata structures that pointed to the old versions of these blocks are updated to point to the new blocks. The file system can then remove the old pointers and the old blocks and make them available for reuse. If the old pointers and blocks are kept, a **snapshot** is created; the snapshot is a view of the file system at a specific point in time (before any updates after that time were applied). This solution should require no consistency checking if the pointer update is done atomically. WAFL does have a consistency checker, however, so some failure scenarios can still cause metadata corruption. (See Section 14.8 for details of the WAFL file system.)

ZFS takes an even more innovative approach to disk consistency. Like WAFL, it never overwrites blocks. However, ZFS goes further and provides checksumming of all metadata and data blocks. This solution (when combined with RAID) assures that data are always correct. ZFS therefore has no consistency checker. (More details on ZFS are found in Section 11.8.6.)

14.7.4 Backup and Restore

Storage devices sometimes fail, and care must be taken to ensure that the data lost in such a failure are not lost forever. To this end, system programs can be used to **back up** data from one storage device to another, such as a magnetic tape or other secondary storage device. Recovery from the loss of an individual file, or of an entire device, may then be a matter of **restoring** the data from backup.

To minimize the copying needed, we can use information from each file's directory entry. For instance, if the backup program knows when the last backup of a file was done, and the file's last write date in the directory indicates that the file has not changed since that date, then the file does not need to be copied again. A typical backup schedule may then be as follows:

- **Day 1**. Copy to a backup medium all files from the file system. This is called a **full backup**.
- **Day 2**. Copy to another medium all files changed since day 1. This is an **incremental backup**.
- **Day 3**. Copy to another medium all files changed since day 2.

. . .

- **Day** N. Copy to another medium all files changed since day $N-1$. Then go back to day 1.

The new cycle can have its backup written over the previous set or onto a new set of backup media.

Using this method, we can restore an entire file system by starting restores with the full backup and continuing through each of the incremental backups. Of course, the larger the value of N, the greater the number of media that must be read for a complete restore. An added advantage of this backup cycle is that we can restore any file accidentally deleted during the cycle by retrieving the deleted file from the backup of the previous day.

The length of the cycle is a compromise between the amount of backup needed and the number of days covered by a restore. To decrease the number of tapes that must be read to do a restore, an option is to perform a full backup and then each day back up all files that have changed since the full backup. In this way, a restore can be done via the most recent incremental backup and the full backup, with no other incremental backups needed. The trade-off is that more files will be modified each day, so each successive incremental backup involves more files and more backup media.

A user may notice that a particular file is missing or corrupted long after the damage was done. For this reason, we usually plan to take a full backup from time to time that will be saved "forever." It is a good idea to store these permanent backups far away from the regular backups to protect against hazard, such as a fire that destroys the computer and all the backups too. In the TV show "Mr. Robot," hackers not only attacked the primary sources of banks' data but also their backup sites. Having multiple backup sites might not be a bad idea if your data are important.

14.8 Example: The WAFL File System

Because secondary-storage I/O has such a huge impact on system performance, file-system design and implementation command quite a lot of attention from system designers. Some file systems are general purpose, in that they can provide reasonable performance and functionality for a wide variety of file sizes, file types, and I/O loads. Others are optimized for specific tasks in an attempt to provide better performance in those areas than general-purpose file systems. The **write-anywhere file layout** (WAFL) from NetApp, Inc. is an example of this sort of optimization. WAFL is a powerful, elegant file system optimized for random writes.

WAFL is used exclusively on network file servers produced by NetApp and is meant for use as a distributed file system. It can provide files to clients via the NFS, CIFS, iSCSI, `ftp`, and `http` protocols, although it was designed just for NFS and CIFS. When many clients use these protocols to talk to a file server, the server may see a very large demand for random reads and an even larger demand for random writes. The NFS and CIFS protocols cache data from read operations, so writes are of the greatest concern to file-server creators.

WAFL is used on file servers that include an NVRAM cache for writes. The WAFL designers took advantage of running on a specific architecture to optimize the file system for random I/O, with a stable-storage cache in front. Ease of use is one of the guiding principles of WAFL. Its creators also designed it to include a new snapshot functionality that creates multiple read-only copies of the file system at different points in time, as we shall see.

The file system is similar to the Berkeley Fast File System, with many modifications. It is block-based and uses inodes to describe files. Each inode contains 16 pointers to blocks (or indirect blocks) belonging to the file described by the inode. Each file system has a root inode. All of the metadata lives in files. All inodes are in one file, the free-block map in another, and the free-inode map in a third, as shown in Figure 14.12. Because these are standard files, the data blocks are not limited in location and can be placed anywhere. If a file system is expanded by addition of disks, the lengths of the metadata files are automatically expanded by the file system.

Thus, a WAFL file system is a tree of blocks with the root inode as its base. To take a snapshot, WAFL creates a copy of the root inode. Any file or metadata updates after that go to new blocks rather than overwriting their existing blocks. The new root inode points to metadata and data changed as a result of these writes. Meanwhile, the snapshot (the old root inode) still points to the old blocks, which have not been updated. It therefore provides access to the file system just as it was at the instant the snapshot was made—and takes very little storage space to do so. In essence, the extra space occupied by a snapshot consists of just the blocks that have been modified since the snapshot was taken.

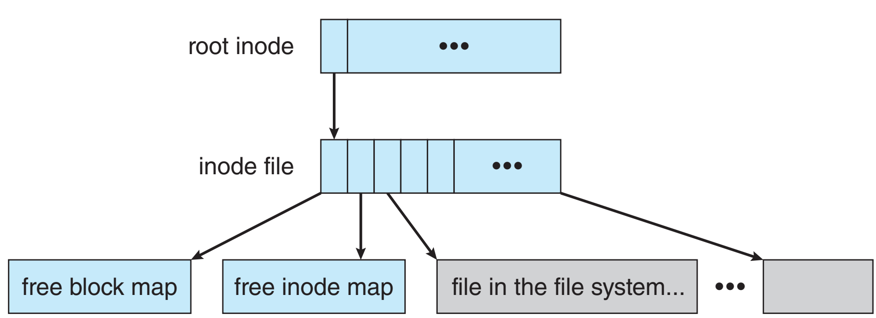
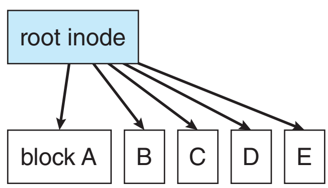


Figure 14.12 The WAFL file layout.

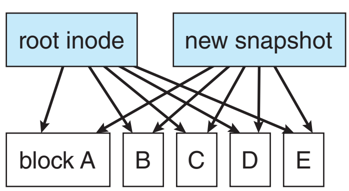
An important change from more standard file systems is that the free-block map has more than one bit per block. It is a bitmap with a bit set for each snapshot that is using the block. When all snapshots that have been using the block are deleted, the bitmap for that block is all zeros, and the block is free to be reused. Used blocks are never overwritten, so writes are very fast, because a write can occur at the free block nearest the current head location. There are many other performance optimizations in WAFL as well.

Many snapshots can exist simultaneously, so one can be taken each hour of the day and each day of the month, for example. A user with access to these snapshots can access files as they were at any of the times the snapshots were taken. The snapshot facility is also useful for backups, testing, versioning, and so on. WAFL's snapshot facility is very efficient in that it does not even require that copy-on-write copies of each data block be taken before the block is modified. Other file systems provide snapshots, but frequently with less efficiency. WAFL snapshots are depicted in Figure 14.13.

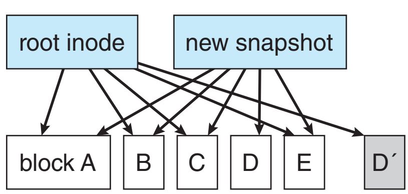
Newer versions of WAFL actually allow read–write snapshots, known as **clones**. Clones are also efficient, using the same techniques as shapshots. In this case, a read-only snapshot captures the state of the file system, and a clone refers back to that read-only snapshot. Any writes to the clone are stored in new blocks, and the clone's pointers are updated to refer to the new blocks. The original snapshot is unmodified, still giving a view into the file system as



(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D´.

Figure 14.13 Snapshots in WAFL.

THE APPLE FILE SYSTEM

In 2017, Apple, Inc., released a new file system to replace its 30-year-old HFS+ file system. HFS+ had been stretched to add many new features, but as usual, this process added complexity, along with lines of code, and made adding more features more difficult. Starting from scratch on a blank page allows a design to start with current technologies and methodologies and provide the exact set of features needed.

Apple File System (**APFS**) is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, I/Os, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit pointers, clones for files and directories, snapshots, space sharing, fast directory sizing, atomic safe-save primitives, copy-on-write design, encryption (single- and multi-key), and I/O coalescing. It understands NVM as well as HDD storage.

Most of these features we've discussed, but there are a few new concepts worth exploring. **Space sharing** is a ZFS-like feature in which storage is available as one or more large free spaces (**containers**) from which file systems can draw allocations (allowing APFS-formatted volumes to grow and shrink). **Fast directory sizing** provides quick used-space calculation and updating. **Atomic safe-save** is a primitive (available via API, not via file-system commands) that performs renames of files, bundles of files, and directories as single atomic operations. I/O coalescing is an optimization for NVM devices in which several small writes are gathered together into a large write to optimize write performance.

Apple chose not to implement RAID as part of the new APFS, instead depending on the existing Apple RAID volume mechanism for software RAID. APFS is also compatible with HFS+, allowing easy conversion for existing deployments.

it was before the clone was updated. Clones can also be promoted to replace the original file system; this involves throwing out all of the old pointers and any associated old blocks. Clones are useful for testing and upgrades, as the original version is left untouched and the clone deleted when the test is done or if the upgrade fails.

Another feature that naturally results from the WAFL file system implementation is **replication,** the duplication and synchronization of a set of data over a network to another system. First, a snapshot of a WAFL file system is duplicated to another system. When another snapshot is taken on the source system, it is relatively easy to update the remote system just by sending over all blocks contained in the new snapshot. These blocks are the ones that have changed between the times the two snapshots were taken. The remote system adds these blocks to the file system and updates its pointers, and the new system then is a duplicate of the source system as of the time of the second snapshot. Repeating this process maintains the remote system as a nearly up-to-date copy of the first system. Such replication is used for disaster recovery. Should the first system be destroyed, most of its data are available for use on the remote system.

Finally, note that the ZFS file system supports similarly efficient snapshots, clones, and replication, and those features are becoming more common in various file systems as time goes by.

14.9 Summary

- Most file systems reside on secondary storage, which is designed to hold a large amount of data permanently. The most common secondary-storage medium is the disk, but the use of NVM devices is increasing.
- Storage devices are segmented into partitions to control media use and to allow multiple, possibly varying, file systems on a single device. These file systems are mounted onto a logical file system architecture to make them available for use.
- File systems are often implemented in a layered or modular structure. The lower levels deal with the physical properties of storage devices and communicating with them. Upper levels deal with symbolic file names and logical properties of files.
- The various files within a file system can be allocated space on the storage device in three ways: through contiguous, linked, or indexed allocation. Contiguous allocation can suffer from external fragmentation. Direct access is very inefficient with linked allocation. Indexed allocation may require substantial overhead for its index block. These algorithms can be optimized in many ways. Contiguous space can be enlarged through extents to increase flexibility and to decrease external fragmentation. Indexed allocation can be done in clusters of multiple blocks to increase throughput and to reduce the number of index entries needed. Indexing in large clusters is similar to contiguous allocation with extents.
- Free-space allocation methods also influence the efficiency of disk-space use, the performance of the file system, and the reliability of secondary storage. The methods used include bit vectors and linked lists. Optimizations include grouping, counting, and the FAT, which places the linked list in one contiguous area.
- Directory-management routines must consider efficiency, performance, and reliability. A hash table is a commonly used method, as it is fast and efficient. Unfortunately, damage to the table or a system crash can result in inconsistency between the directory information and the disk's contents.
- A consistency checker can be used to repair damaged file-system structures. Operating-system backup tools allow data to be copied to magnetic tape or other storage devices, enabling the user to recover from data loss or even entire device loss due to hardware failure, operating system bug, or user error.
- Due to the fundamental role that file systems play in system operation, their performance and reliability are crucial. Techniques such as log structures and caching help improve performance, while log structures and RAID improve reliability. The WAFL file system is an example of optimization of performance to match a specific I/O load.

Practice Exercises

14.1 Consider a file currently consisting of 100 blocks. Assume that the file-control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow at the beginning but there is room to grow at the end. Also assume that the block information to be added is stored in memory.

a. The block is added at the beginning.

b. The block is added in the middle.

c. The block is added at the end.

d. The block is removed from the beginning.

e. The block is removed from the middle.

f. The block is removed from the end.

14.2 Why must the bit map for file allocation be kept on mass storage, rather than in main memory?

14.3 Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?

14.4 One problem with contiguous allocation is that the user must preallocate enough space for each file. If the file grows to be larger than the space allocated for it, special actions must be taken. One solution to this problem is to define a file structure consisting of an initial contiguous area of a specified size. If this area is filled, the operating system automatically defines an overflow area that is linked to the initial contiguous area. If the overflow area is filled, another overflow area is allocated. Compare this implementation of a file with the standard contiguous and linked implementations.

14.5 How do caches help improve performance? Why do systems not use more or larger caches if they are so useful?

14.6 Why is it advantageous to the user for an operating system to dynamically allocate its internal tables? What are the penalties to the operating system for doing so?

Further Reading

The internals of the BSD UNIX system are covered in full in [McKusick et al. (2015)]. Details concerning file systems for Linux can be found in [Love (2010)]. The Google file system is described in [Ghemawat et al. (2003)]. FUSE can be found at http://fuse.sourceforge.net.

Log-structured file organizations for enhancing both performance and consistency are discussed in [Rosenblum and Ousterhout (1991)], [Seltzer et al. (1993)], and [Seltzer et al. (1995)]. Log-structured designs for networked file systems are proposed in [Hartman and Ousterhout (1995)] and [Thekkath et al. (1997)].

The ZFS source code for space maps can be found at http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/ fs/zfs/space_map.c.

ZFS documentation can be found at http://www.opensolaris.org/os/community/ZFS/docs.

The NTFS file system is explained in [Solomon (1998)], the Ext3 file system used in Linux is described in [Mauerer (2008)], and the WAFL file system is covered in [Hitz et al. (1995)].

Bibliography

[Ghemawat et al. (2003)] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System", *Proceedings of the ACM Symposium on Operating Systems Principles* (2003).

[Hartman and Ousterhout (1995)] J. H. Hartman and J. K. Ousterhout, "The Zebra Striped Network File System", *ACM Transactions on Computer Systems*, Volume 13, Number 3 (1995), pages 274–310.

[Hitz et al. (1995)] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance", Technical report, NetApp (1995).

[Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

[Mauerer (2008)] W. Mauerer, *Professional Linux Kernel Architecture*, John Wiley and Sons (2008).

[McKusick et al. (2015)] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System–Second Edition*, Pearson (2015).

[Rosenblum and Ousterhout (1991)] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", *Proceedings of the ACM Symposium on Operating Systems Principles* (1991), pages 1–15.

[Seltzer et al. (1993)] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX", *USENIX Winter* (1993), pages 307–326.

[Seltzer et al. (1995)] M. I. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. N. Padmanabhan, "File System Logging Versus Clustering: A Performance Comparison", *USENIX Winter* (1995), pages 249–264.

[Solomon (1998)] D. A. Solomon, *Inside Windows NT,* Second Edition, Microsoft Press (1998).

[Thekkath et al. (1997)] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: A Scalable Distributed File System", *Symposium on Operating Systems Principles* (1997), pages 224–237.

Chapter 14 Exercises

14.7 Consider a file system that uses a modified contiguous-allocation scheme with support for extents. A file is a collection of extents, with each extent corresponding to a contiguous set of blocks. A key issue in such systems is the degree of variability in the size of the extents. What are the advantages and disadvantages of the following schemes?

a. All extents are of the same size, and the size is predetermined.

b. Extents can be of any size and are allocated dynamically.

c. Extents can be of a few fixed sizes, and these sizes are predetermined.

14.8 Contrast the performance of the three techniques for allocating disk blocks (contiguous, linked, and indexed) for both sequential and random file access.

14.9 What are the advantages of the variant of linked allocation that uses a FAT to chain together the blocks of a file?

14.10 Consider a system where free space is kept in a free-space list.

a. Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.

b. Consider a file system similar to the one used by UNIX with indexed allocation. How many disk I/O operations might be required to read the contents of a small local file at `/a/b/c`? Assume that none of the disk blocks is currently being cached.

c. Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.

14.11 Some file systems allow disk storage to be allocated at different levels of granularity. For instance, a file system could allocate 4 KB of disk space as a single 4-KB block or as eight 512-byte blocks. How could we take advantage of this flexibility to improve performance? What modifications would have to be made to the free-space management scheme in order to support this feature?

14.12 Discuss how performance optimizations for file systems might result in difficulties in maintaining the consistency of the systems in the event of computer crashes.

14.13 Discuss the advantages and disadvantages of supporting links to files that cross mount points (that is, the file link refers to a file that is stored in a different volume).

14.14 Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer these questions:

a. How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long.)

b. If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?

14.15 Consider a file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

14.16 Fragmentation on a storage device can be eliminated through compaction. Typical disk devices do not have relocation or base registers (such as those used when memory is to be compacted), so how can we relocate files? Give three reasons why compacting and relocating files are often avoided.

14.17 Explain why logging metadata updates ensures recovery of a file system after a file-system crash.

14.18 Consider the following backup scheme:

- **Day 1**. Copy to a backup medium all files from the disk.
- **Day 2**. Copy to another medium all files changed since day 1.
- **Day 3**. Copy to another medium all files changed since day 1.

This differs from the schedule given in Section 14.7.4 by having all subsequent backups copy all files modified since the first full backup. What are the benefits of this system over the one in Section 14.7.4? What are the drawbacks? Are restore operations made easier or more difficult? Explain your answer.

14.19 Discuss the advantages and disadvantages of associating with remote file systems (stored on file servers) a set of failure semantics different from those associated with local file systems.

14.20 What are the implications of supporting UNIX consistency semantics for shared access to files stored on remote file systems?

CHAPTER 15

File-System Internals

As we saw in Chapter 13, the file system provides the mechanism for on-line storage and access to file contents, including data and programs. This chapter is primarily concerned with the internal structures and operations of file systems. We explore in detail ways to structure file use, to allocate storage space, to recover freed space, to track the locations of data, and to interface other parts of the operating system to secondary storage.

CHAPTER OBJECTIVES

- Delve into the details of file systems and their implementation.
- Explore booting and file sharing.
- Describe remote file systems, using NFS as an example.

15.1 File Systems

Certainly, no general-purpose computer stores just one file. There are typically thousands, millions, even billions of files within a computer. Files are stored on random-access storage devices, including hard disk drives, optical disks, and nonvolatile memory devices.

As you have seen in the preceding chapters, a general-purpose computer system can have multiple storage devices, and those devices can be sliced up into partitions, which hold volumes, which in turn hold file systems. Depending on the volume manager, a volume may span multiple partitions as well. Figure 15.1 shows a typical file-system organization.

Computer systems may also have varying numbers of file systems, and the file systems may be of varying types. For example, a typical Solaris system may have dozens of file systems of a dozen different types, as shown in the file-system list in Figure 15.2.

In this book, we consider only general-purpose file systems. It is worth noting, though, that there are many special-purpose file systems. Consider the types of file systems in the Solaris example mentioned above:

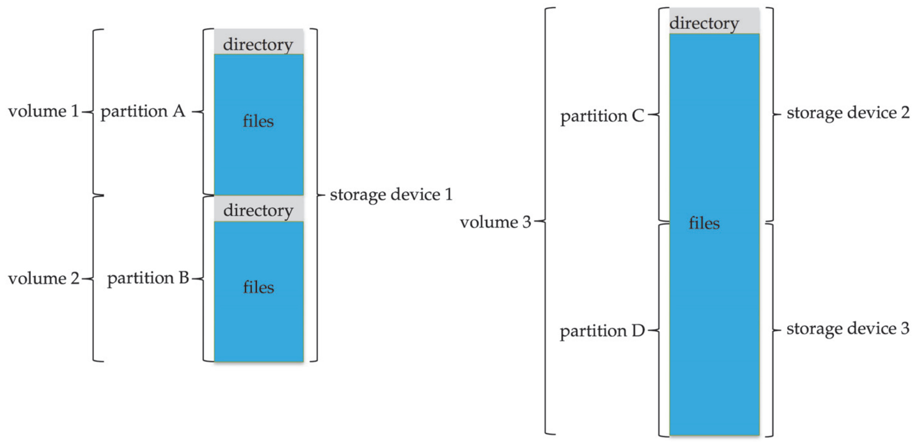


Figure 15.1 A typical storage device organization.

- **tmpfs**—a "temporary" file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- **objfs**—a "virtual" file system (essentially an interface to the kernel that looks like a file system) that gives debuggers access to kernel symbols
- **ctfs**—a virtual file system that maintains "contract" information to manage which processes start when the system boots and must continue to run during operation
- **lofs**—a "loop back" file system that allows one file system to be accessed in place of another one
- **procfs**—a virtual file system that presents information on all processes as a file system
- **ufs, zfs**—general-purpose file systems

The file systems of computers, then, can be extensive. Even within a file system, it is useful to segregate files into groups and manage and act on those groups. This organization involves the use of directories (see Section 14.3).

15.2 File-System Mounting

Just as a file must be opened before it can be used, a file system must be mounted before it can be available to processes on the system. More specifically, the directory structure may be built out of multiple file-system-containing volumes, which must be mounted to make them available within the file-system name space.

The mount procedure is straightforward. The operating system is given the name of the device and the **mount point**—the location within the file structure where the file system is to be attached. Some operating systems require that a file-system type be provided, while others inspect the structures of the device

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

Figure 15.2 Solaris file systems.

and determine the type of file system. Typically, a mount point is an empty directory. For instance, on a UNIX system, a file system containing a user's home directories might be mounted as `/home`; then, to access the directory structure within that file system, we could precede the directory names with `/home`, as in `/home/jane`. Mounting that file system under `/users` would result in the path name `/users/jane`, which we could use to reach the same directory.

Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems, and even file systems of varying types, as appropriate.

To illustrate file mounting, consider the file system depicted in Figure 15.3, where the triangles represent subtrees of directories that are of interest. Figure 15.3(a) shows an existing file system, while Figure 15.3(b) shows an unmounted volume residing on `/device/dsk`. At this point, only the files on the existing file system can be accessed. Figure 15.4 shows the effects of mounting the volume residing on `/device/dsk` over `/users`. If the volume is unmounted, the file system is restored to the situation depicted in Figure 15.3.

Systems impose semantics to clarify functionality. For example, a system may disallow a mount over a directory that contains files; or it may make the

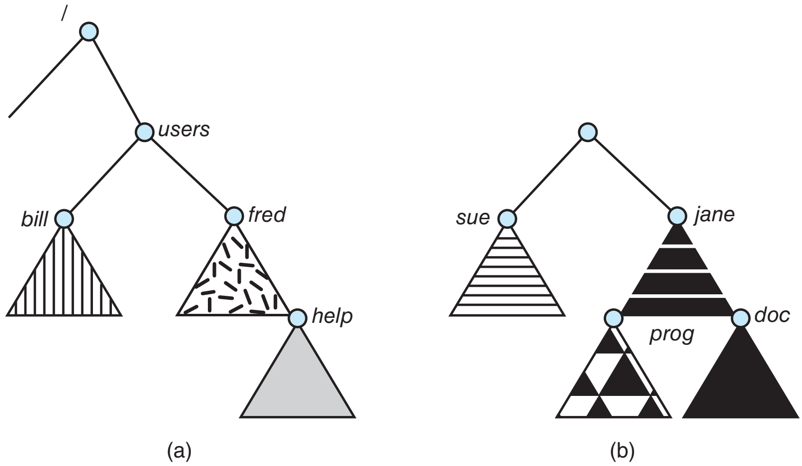


Figure 15.3 File system. (a) Existing system. (b) Unmounted volume.

mounted file system available at that directory and obscure the directory's existing files until the file system is unmounted, terminating the use of the file system and allowing access to the original files in that directory. As another example, a system may allow the same file system to be mounted repeatedly, at different mount points; or it may only allow one mount per file system.

Consider the actions of the macOS operating system. Whenever the system encounters a disk for the first time (either at boot time or while the system is running), the macOS operating system searches for a file system on the device. If it finds one, it automatically mounts the file system under the `/Volumes` directory, adding a folder icon labeled with the name of the file system (as stored in the device directory). The user is then able to click on the icon and thus display the newly mounted file system.

The Microsoft Windows family of operating systems maintains an extended two-level directory structure, with devices and volumes assigned drive letters. Each volume has a general graph directory structure associated

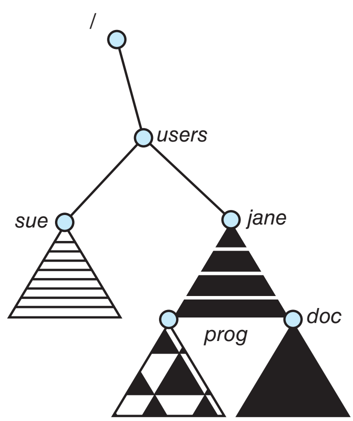


Figure 15.4 Volume mounted at /users.

with its drive letter. The path to a specific file takes the form `drive-letter:\path\to\file`. The more recent versions of Windows allow a file system to be mounted anywhere in the directory tree, just as UNIX does. Windows operating systems automatically discover all devices and mount all located file systems at boot time. In some systems, like UNIX, the mount commands are explicit. A system configuration file contains a list of devices and mount points for automatic mounting at boot time, but other mounts may be executed manually.

Issues concerning file system mounting are further discussed in Section 15.3 and in Section C.7.5.

15.3 Partitions and Mounting

The layout of a disk can have many variations, depending on the operating system and volume management software. A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks. The former layout is discussed here, while the latter, which is more appropriately considered a form of RAID, is covered in Section 11.8.

Each partition can be either "raw," containing no file system, or "cooked," containing a file system. **Raw disk** is used where no file system is appropriate. UNIX swap space can use a raw partition, for example, since it uses its own format on disk and does not use a file system. Likewise, some databases use raw disk and format the data to suit their needs. Raw disk can also hold information needed by disk RAID systems, such as bitmaps indicating which blocks are mirrored and which have changed and need to be mirrored. Similarly, raw disk can contain a miniature database holding RAID configuration information, such as which disks are members of each RAID set. Raw disk use is discussed in Section 11.5.1.

If a partition contains a file system that is bootable—that has a properly installed and configured operating system—then the partition also needs boot information, as described in Section 11.5.2. This information has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format. Rather, boot information is usually a sequential series of blocks loaded as an image into memory. Execution of the image starts at a predefined location, such as the first byte. This image, the **bootstrap loader**, in turn knows enough about the file-system structure to be able to find and load the kernel and start it executing.

The boot loader can contain more than the instructions for booting a specific operating system. For instance, many systems can be **dual-booted**, allowing us to install multiple operating systems on a single system. How does the system know which one to boot? A boot loader that understands multiple file systems and multiple operating systems can occupy the boot space. Once loaded, it can boot one of the operating systems available on the drive. The drive can have multiple partitions, each containing a different type of file system and a different operating system. Note that if the boot loader does not understand a particular file-system format, an operating system stored on that file system is not bootable. This is one of the reasons only some file systems are supported as root file systems for any given operating system.

The **root partition** selected by the boot loader, which contains the operating-system kernel and sometimes other system files, is mounted at boot time. Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system. As part of a successful mount operation, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention. Finally, the operating system notes in its in-memory mount table that a file system is mounted, along with the type of the file system. The details of this function depend on the operating system.

Microsoft Windows–based systems mount each volume in a separate name space, denoted by a letter and a colon, as mentioned earlier. To record that a file system is mounted at `F:`, for example, the operating system places a pointer to the file system in a field of the device structure corresponding to `F:`. When a process specifies the driver letter, the operating system finds the appropriate file-system pointer and traverses the directory structures on that device to find the specified file or directory. Later versions of Windows can mount a file system at any point within the existing directory structure.

On UNIX, file systems can be mounted at any directory. Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory. The flag indicates that the directory is a mount point. A field then points to an entry in the mount table, indicating which device is mounted there. The mount table entry contains a pointer to the superblock of the file system on that device. This scheme enables the operating system to traverse its directory structure, switching seamlessly among file systems of varying types.

15.4 File Sharing

The ability to share files is very desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal. Therefore, user-oriented operating systems must accommodate the need to share files in spite of the inherent difficulties.

In this section, we examine more aspects of file sharing. We begin by discussing general issues that arise when multiple users share files. Once multiple users are allowed to share files, the challenge is to extend sharing to multiple file systems, including remote file systems; we discuss that challenge as well. Finally, we consider what to do about conflicting actions occurring on shared files. For instance, if multiple users are writing to a file, should all the writes be allowed to occur, or should the operating system protect the users' actions from one another?

15.4.1 Multiple Users

When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent. Given a directory structure that allows files to be shared by users, the system must mediate the file sharing. The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files. These are the issues of access control and protection, which are covered in Section 13.4.

To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system. Although many approaches have been taken to meet this requirement, most systems have evolved to use the concepts of file (or directory) **owner** (or **user**) and **group**. The owner is the user who can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file. For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations. Exactly which operations can be executed by group members and other users is definable by the file's owner.

The owner and group IDs of a given file (or directory) are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file. Likewise, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

Many systems have multiple local file systems, including volumes of a single disk or multiple volumes on multiple attached disks. In these cases, the ID checking and permission matching are straightforward, once the file systems are mounted. But consider an external disk that can be moved between systems. What if the IDs on the systems are different? Care must be taken to be sure that IDs match between systems when devices move between them or that file ownership is reset when such a move occurs. (For example, we can create a new user ID and set all files on the portable disk to that ID, to be sure no files are accidentally accessible to existing users.)

15.5 Virtual File Systems

As we've seen, modern operating systems must concurrently support multiple types of file systems. But how does an operating system allow multiple types of file systems to be integrated into a directory structure? And how can users seamlessly move between file-system types as they navigate the file-system space? We now discuss some of these implementation details.

An obvious but suboptimal method of implementing multiple types of file systems is to write directory and file routines for each type. Instead, however, most operating systems, including UNIX, use object-oriented techniques to simplify, organize, and modularize the implementation. The use of these methods allows very dissimilar file-system types to be implemented within the same structure, including network file systems, such as NFS. Users can access files contained within multiple file systems on the local drive or even on file systems available across the network.

Data structures and procedures are used to isolate the basic system-call functionality from the implementation details. Thus, the file-system implementation consists of three major layers, as depicted schematically in Figure 15.5. The first layer is the file-system interface, based on the `open()`, `read()`, `write()`, and `close()` calls and on file descriptors.

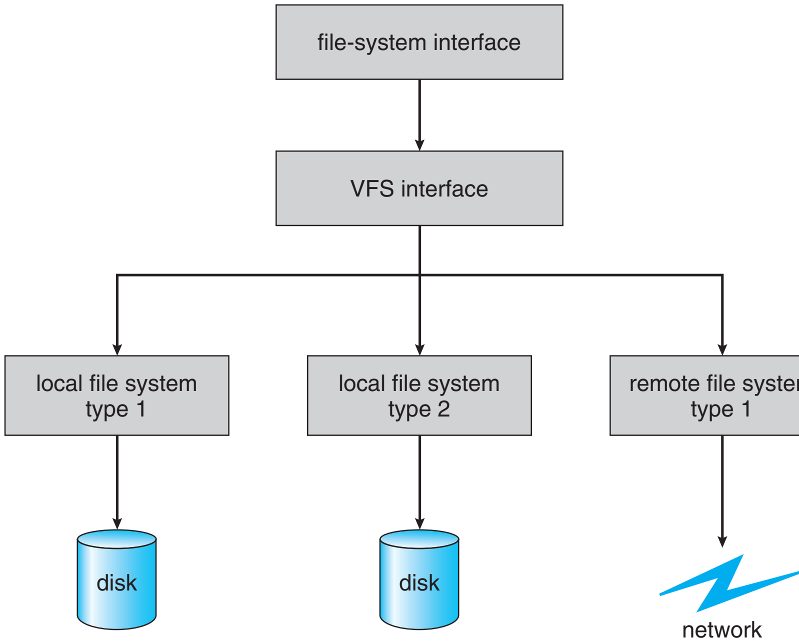


Figure 15.5 Schematic view of a virtual file system.

The second layer is called the **virtual file system** (**VFS**) layer. The VFS layer serves two important functions:

1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
2. It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a **vnode**, that contains a numerical designator for a network-wide unique file. (UNIX inodes are unique within only a single file system.) This network-wide uniqueness is required for support of network file systems. The kernel maintains one vnode structure for each active node (file or directory).

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.

The VFS activates file-system-specific operations to handle local requests according to their file-system types and calls the NFS protocol procedures (or other protocol procedures for other network file systems) for remote requests. File handles are constructed from the relevant vnodes and are passed as arguments to these procedures. The layer implementing the file-system type or the remote-file-system protocol is the third layer of the architecture.

Let's briefly examine the VFS architecture in Linux. The four main object types defined by the Linux VFS are:

- The **inode object**, which represents an individual file
- The **fil object**, which represents an open file
- The **superblock object**, which represents an entire file system
- The **dentry object**, which represents an individual directory entry

For each of these four object types, the VFS defines a set of operations that may be implemented. Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that particular object. For example, an abbreviated API for some of the operations for the file object includes:

- `int open(. . .)`—Open a file.
- `int close(. . .)`—Close an already-open file.
- `ssize_t read(. . .)`—Read from a file.
- `ssize_t write(. . .)`—Write to a file.
- `int mmap(. . .)`—Memory-map a file.

An implementation of the file object for a specific file type is required to implement each function specified in the definition of the file object. (The complete definition of the file object is specified in the file `struct file_operations`, which is located in the file `/usr/include/linux/fs.h`.)

Thus, the VFS software layer can perform an operation on one of these objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with. The VFS does not know, or care, whether an inode represents a disk file, a directory file, or a remote file. The appropriate function for that file's `read()` operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.

15.6 Remote File Systems

With the advent of networks (Chapter 19), communication among remote computers became possible. Networking allows the sharing of resources spread across a campus or even around the world. One obvious resource to share is data in the form of files.

Through the evolution of network and file technology, remote file-sharing methods have changed. The first implemented method involves manually transferring files between machines via programs like `ftp`. The second major method uses a **distributed fil system** (**DFS**), in which remote directories are visible from a local machine. In some ways, the third method, the **World Wide Web**, is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for `ftp`) are used to transfer files. Increasingly, cloud computing (Section 1.10.5) is being used for file sharing as well.

`ftp` is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote system. The World Wide Web uses anonymous file exchange almost exclusively. DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files. This integration adds complexity, as we describe in this section.

15.6.1 The Client–Server Model

Remote file systems allow a computer to mount one or more file systems from one or more remote machines. In this case, the machine containing the files is the **server**, and the machine seeking access to the files is the **client**. The client–server relationship is common with networked machines. Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client–server facility.

The server usually specifies the available files on a volume or directory level. Client identification is more difficult. A client can be specified by a network name or other identifier, such as an IP address, but these can be **spoofed**, or imitated. As a result of spoofing, an unauthorized client could be allowed access to the server. More secure solutions include secure authentication of the client via encrypted keys. Unfortunately, with security come many challenges, including ensuring compatibility of the client and server (they must use the same encryption algorithms) and security of key exchanges (intercepted keys could again allow unauthorized access). Because of the difficulty of solving these problems, unsecure authentication methods are most commonly used.

In the case of UNIX and its network file system (NFS), authentication takes place via the client networking information, by default. In this scheme, the user's IDs on the client and server must match. If they do not, the server will be unable to determine access rights to files. Consider the example of a user who has an ID of 1000 on the client and 2000 on the server. A request from the client to the server for a specific file will not be handled appropriately, as the server will determine if user 1000 has access to the file rather than basing the determination on the real user ID of 2000. Access is thus granted or denied based on incorrect authentication information. The server must trust the client to present the correct user ID. Note that the NFS protocols allow many-to-many relationships. That is, many servers can provide files to many clients. In fact, a given machine can be both a server to some NFS clients and a client of other NFS servers.

Once the remote file system is mounted, file operation requests are sent on behalf of the user across the network to the server via the DFS protocol. Typically, a file-open request is sent along with the ID of the requesting user. The server then applies the standard access checks to determine if the user has credentials to access the file in the mode requested. The request is either allowed or denied. If it is allowed, a file handle is returned to the client application, and the application then can perform read, write, and other operations on the file. The client closes the file when access is completed. The operating system may apply semantics similar to those for a local file-system mount or may use different semantics.

15.6.2 Distributed Information Systems

To make client–server systems easier to manage, **distributed information systems**, also known as **distributed naming services**, provide unified access to the information needed for remote computing. The **domain name system** (**DNS**) provides host-name-to-network-address translations for the entire Internet. Before DNS became widespread, files containing the same information were sent via e-mail or `ftp` between all networked hosts. Obviously, this methodology was not scalable! DNS is further discussed in Section 19.3.1.

Other distributed information systems provide ***user name/password/user ID/group ID*** space for a distributed facility. UNIX systems have employed a wide variety of distributed information methods. Sun Microsystems (now part of Oracle Corporation) introduced **yellow pages** (since renamed **network information service**, or **NIS**), and most of the industry adopted its use. It centralizes storage of user names, host names, printer information, and the like. Unfortunately, it uses unsecure authentication methods, including sending user passwords unencrypted (in clear text) and identifying hosts by IP address. Sun's NIS+ was a much more secure replacement for NIS but was much more complicated and was not widely adopted.

In the case of Microsoft's **common Internet file system** (**CIFS**), network information is used in conjunction with user authentication (user name and password) to create a network login that the server uses to decide whether to allow or deny access to a requested file system. For this authentication to be valid, the user names must match from machine to machine (as with NFS). Microsoft uses **active directory** as a distributed naming structure to provide a single name space for users. Once established, the distributed naming facility is used by all clients and servers to authenticate users via Microsoft's version of the **Kerberos** network authentication protocol (https://web.mit.edu/kerberos/).

The industry is moving toward use of the **lightweight directory-access protocol** (**LDAP**) as a secure distributed naming mechanism. In fact, active directory is based on LDAP. Oracle Solaris and most other major operating systems include LDAP and allow it to be employed for user authentication as well as system-wide retrieval of information, such as availability of printers. Conceivably, one distributed LDAP directory could be used by an organization to store all user and resource information for all the organization's computers. The result would be secure single sign-on for users, who would enter their authentication information once for access to all computers within the organization. It would also ease system-administration efforts by combining, in one location, information that is currently scattered in various files on each system or in different distributed information services.

15.6.3 Failure Modes

Local file systems can fail for a variety of reasons, including failure of the drive containing the file system, corruption of the directory structure or other disk-management information (collectively called **metadata**), disk-controller failure, cable failure, and host-adapter failure. User or system-administrator failure can also cause files to be lost or entire directories or volumes to be deleted. Many of these failures will cause a host to crash and an error condition to be displayed, and human intervention may be required to repair the damage.

Remote file systems have even more failure modes. Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems. In the case of networks, the network can be interrupted between two hosts. Such interruptions can result from hardware failure, poor hardware configuration, or networking implementation issues. Although some networks have built-in resiliency, including multiple paths between hosts, many do not. Any single failure can thus interrupt the flow of DFS commands.

Consider a client in the midst of using a remote file system. It has files open from the remote host; among other activities, it may be performing directory lookups to open files, reading or writing data to files, and closing files. Now consider a partitioning of the network, a crash of the server, or even a scheduled shutdown of the server. Suddenly, the remote file system is no longer reachable. This scenario is rather common, so it would not be appropriate for the client system to act as it would if a local file system were lost. Rather, the system can either terminate all operations to the lost server or delay operations until the server is again reachable. These failure semantics are defined and implemented as part of the remote-file-system protocol. Termination of all operations can result in users' losing data—and patience. Thus, most DFS protocols either enforce or allow delaying of file-system operations to remote hosts, with the hope that the remote host will become available again.

To implement this kind of recovery from failure, some kind of **state information** may be maintained on both the client and the server. If both server and client maintain knowledge of their current activities and open files, then they can seamlessly recover from a failure. In the situation where the server crashes but must recognize that it has remotely mounted exported file systems and opened files, NFS Version 3 takes a simple approach, implementing a **stateless** DFS. In essence, it assumes that a client request for a file read or write would not have occurred unless the file system had been remotely mounted and the file had been previously open. The NFS protocol carries all the information needed to locate the appropriate file and perform the requested operation. Similarly, it does not track which clients have the exported volumes mounted, again assuming that if a request comes in, it must be legitimate. While this stateless approach makes NFS resilient and rather easy to implement, it also makes it unsecure. For example, forged read or write requests could be allowed by an NFS server. These issues are addressed in the industry standard NFS Version 4, in which NFS is made stateful to improve its security, performance, and functionality.

15.7 Consistency Semantics

Consistency semantics represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously. In particular, they specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.

Consistency semantics are directly related to the process synchronization algorithms of Chapter 6. However, the complex algorithms of that chapter tend not to be implemented in the case of file I/O because of the great latencies and slow transfer rates of disks and networks. For example, performing an atomic transaction to a remote disk could involve several network communications, several disk reads and writes, or both. Systems that attempt such a full set of functionalities tend to perform poorly. A successful implementation of complex sharing semantics can be found in the Andrew file system.

For the following discussion, we assume that a series of file accesses (that is, reads and writes) attempted by a user to the same file is always enclosed between the `open()` and `close()` operations. The series of accesses between the `open()` and `close()` operations makes up a **fil session**. To illustrate the concept, we sketch several prominent examples of consistency semantics.

15.7.1 UNIX Semantics

The UNIX file system (Chapter 19) uses the following consistency semantics:

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

In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image causes delays in user processes.

15.7.2 Session Semantics

The Andrew file system (OpenAFS) uses the following consistency semantics:

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

According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay. Almost no constraints are enforced on scheduling accesses.

15.7.3 Immutable-Shared-Files Semantics

A unique approach is that of **immutable shared files**. Once a file is declared as shared by its creator, it cannot be modified. An immutable file has two key properties: its name may not be reused, and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed. The implementation of these semantics in a distributed system (Chapter 19) is simple, because the sharing is disciplined (read-only).

15.8 NFS

Network file systems are commonplace. They are typically integrated with the overall directory structure and interface of the client system. NFS is a good example of a widely used, well implemented client–server network file system. Here, we use it as an example to explore the implementation details of network file systems.

NFS is both an implementation and a specification of a software system for accessing remote files across LANs (or even WANs). NFS is part of ONC+, which most UNIX vendors and some PC operating systems support. The implementation described here is part of the Solaris operating system, which is a modified version of UNIX SVR4. It uses either the TCP or UDP/IP protocol (depending on the interconnecting network). The specification and the implementation are intertwined in our description of NFS. Whenever detail is needed, we refer to the Solaris implementation; whenever the description is general, it applies to the specification also.

There are multiple versions of NFS, with the latest being Version 4. Here, we describe Version 3, which is the version most commonly deployed.

15.8.1 Overview

NFS views a set of interconnected workstations as a set of independent machines with independent file systems. The goal is to allow some degree of sharing among these file systems (on explicit request) in a transparent manner. Sharing is based on a client–server relationship. A machine may be, and often is, both a client and a server. Sharing is allowed between any pair of machines. To ensure machine independence, sharing of a remote file system affects only the client machine and no other machine.

So that a remote directory will be accessible in a transparent manner from a particular machine—say, from ***M1***—a client of that machine must first carry out a mount operation. The semantics of the operation involve mounting a remote directory over a directory of a local file system. Once the mount operation is completed, the mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory. The local directory becomes the name of the root of the newly mounted directory. Specification of the remote directory as an argument for the mount operation is not done transparently; the location (or host name) of the remote directory has to be provided. However, from then on, users on machine ***M1*** can access files in the remote directory in a totally transparent manner.

To illustrate file mounting, consider the file system depicted in Figure 15.6, where the triangles represent subtrees of directories that are of interest. The figure shows three independent file systems of machines named ***U***, ***S1***, and ***S2***. At this point, on each machine, only the local files can be accessed. Figure 15.7(a) shows the effects of mounting `S1:/usr/shared` over `U:/usr/local`. This figure depicts the view users on ***U*** have of their file system. After the mount is complete, they can access any file within the `dir1` directory using the prefix `/usr/local/dir1`. The original directory `/usr/local` on that machine is no longer visible.

Subject to access-rights accreditation, any file system, or any directory within a file system, can be mounted remotely on top of any local directory.

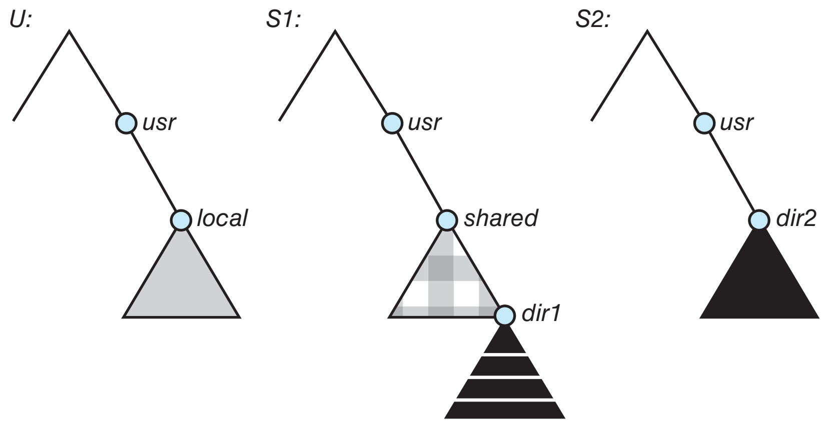


Figure 15.6 Three independent file systems.

Diskless workstations can even mount their own roots from servers. Cascading mounts are also permitted in some NFS implementations. That is, a file system can be mounted over another file system that is remotely mounted, not local. A machine is affected by only those mounts that it has itself invoked. Mounting a remote file system does not give the client access to other file systems that were, by chance, mounted over the former file system. Thus, the mount mechanism does not exhibit a transitivity property.

In Figure 15.7(b), we illustrate cascading mounts. The figure shows the result of mounting `S2:/usr/dir2` over `U:/usr/local/dir1`, which is already remotely mounted from ***S1***. Users can access files within `dir2` on ***U*** using the prefix `/usr/local/dir1`. If a shared file system is mounted over a user's home directories on all machines in a network, the user can log into any workstation and get his or her home environment. This property permits user mobility.

One of the design goals of NFS was to operate in a heterogeneous environment of different machines, operating systems, and network architectures. The

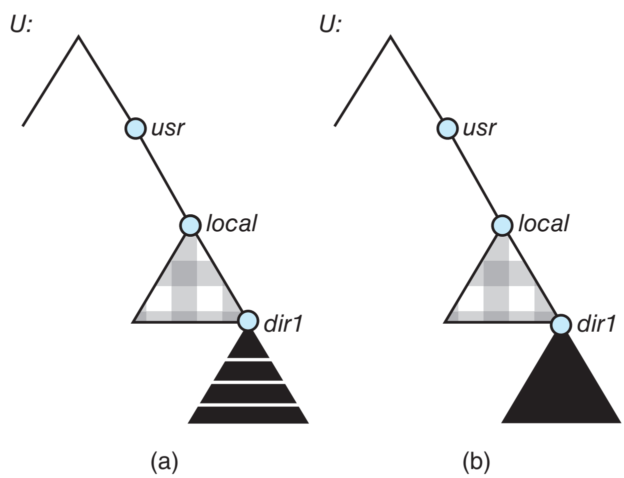


Figure 15.7 Mounting in NFS. (a) Mounts. (b) Cascading mounts.

NFS specification is independent of these media. This independence is achieved through the use of RPC primitives built on top of an external data representation (XDR) protocol used between two implementation-independent interfaces. Hence, if the system's heterogeneous machines and file systems are properly interfaced to NFS, file systems of different types can be mounted both locally and remotely.

The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services. Accordingly, two separate protocols are specified for these services: a mount protocol and a protocol for remote file accesses, the **NFS protocol**. The protocols are specified as sets of RPCs. These RPCs are the building blocks used to implement transparent remote file access.

15.8.2 The Mount Protocol

The **mount protocol** establishes the initial logical connection between a server and a client. In Solaris, each machine has a server process, outside the kernel, performing the protocol functions.

A mount operation includes the name of the remote directory to be mounted and the name of the server machine storing it. The mount request is mapped to the corresponding RPC and is forwarded to the mount server running on the specific server machine. The server maintains an **export list** that specifies local file systems that it exports for mounting, along with names of machines that are permitted to mount them. (In Solaris, this list is the `/etc/dfs/dfstab`, which can be edited only by a superuser.) The specification can also include access rights, such as read only. To simplify the maintenance of export lists and mount tables, a distributed naming scheme can be used to hold this information and make it available to appropriate clients.

Recall that any directory within an exported file system can be mounted remotely by an accredited machine. A component unit is such a directory. When the server receives a mount request that conforms to its export list, it returns to the client a file handle that serves as the key for further accesses to files within the mounted file system. The file handle contains all the information that the server needs to distinguish an individual file it stores. In UNIX terms, the file handle consists of a file-system identifier and an inode number to identify the exact mounted directory within the exported file system.

The server also maintains a list of the client machines and the corresponding currently mounted directories. This list is used mainly for administrative purposes—for instance, for notifying all clients that the server is going down. Only through addition and deletion of entries in this list can the server state be affected by the mount protocol.

Usually, a system has a static mounting preconfiguration that is established at boot time (`/etc/vfstab` in Solaris); however, this layout can be modified. In addition to the actual mount procedure, the mount protocol includes several other procedures, such as unmount and return export list.

15.8.3 The NFS Protocol

The NFS protocol provides a set of RPCs for remote file operations. The procedures support the following operations:

- Searching for a file within a directory
- Reading a set of directory entries
- Manipulating links and directories
- Accessing file attributes
- Reading and writing files

These procedures can be invoked only after a file handle for the remotely mounted directory has been established.

The omission of open and close operations is intentional. A prominent feature of NFS servers is that they are stateless. Servers do not maintain information about their clients from one access to another. No parallels to UNIX's open-files table or file structures exist on the server side. Consequently, each request has to provide a full set of arguments, including a unique file identifier and an absolute offset inside the file for the appropriate operations. The resulting design is robust; no special measures need be taken to recover a server after a crash. File operations must be idempotent for this purpose—that is, the same operation performed multiple times must have the same effect as if it had only been performed once. To achieve idempotence, every NFS request has a sequence number, allowing the server to determine if a request has been duplicated or if any are missing.

Maintaining the list of clients that we mentioned seems to violate the statelessness of the server. However, this list is not essential for the correct operation of the client or the server, and hence it does not need to be restored after a server crash. Consequently, it may include inconsistent data and is treated as only a hint.

A further implication of the stateless-server philosophy and a result of the synchrony of an RPC is that modified data (including indirection and status blocks) must be committed to the server's disk before results are returned to the client. That is, a client can cache write blocks, but when it flushes them to the server, it assumes that they have reached the server's disks. The server must write all NFS data synchronously. Thus, a server crash and recovery will be invisible to a client; all blocks that the server is managing for the client will be intact. The resulting performance penalty can be large, because the advantages of caching are lost. Performance can be increased by using storage with its own nonvolatile cache (usually battery-backed-up memory). The disk controller acknowledges the disk write when the write is stored in the nonvolatile cache. In essence, the host sees a very fast synchronous write. These blocks remain intact even after a system crash and are written from this stable storage to disk periodically.

A single NFS write procedure call is guaranteed to be atomic and is not intermixed with other write calls to the same file. The NFS protocol, however, does not provide concurrency-control mechanisms. A `write()` system call may be broken down into several RPC writes, because each NFS write or read call can contain up to 8 KB of data and UDP packets are limited to 1,500 bytes. As a result, two users writing to the same remote file may get their data intermixed. The claim is that, because lock management is inherently stateful, a service outside the NFS should provide locking (and Solaris does). Users are advised to coordinate access to shared files using mechanisms outside the scope of NFS.

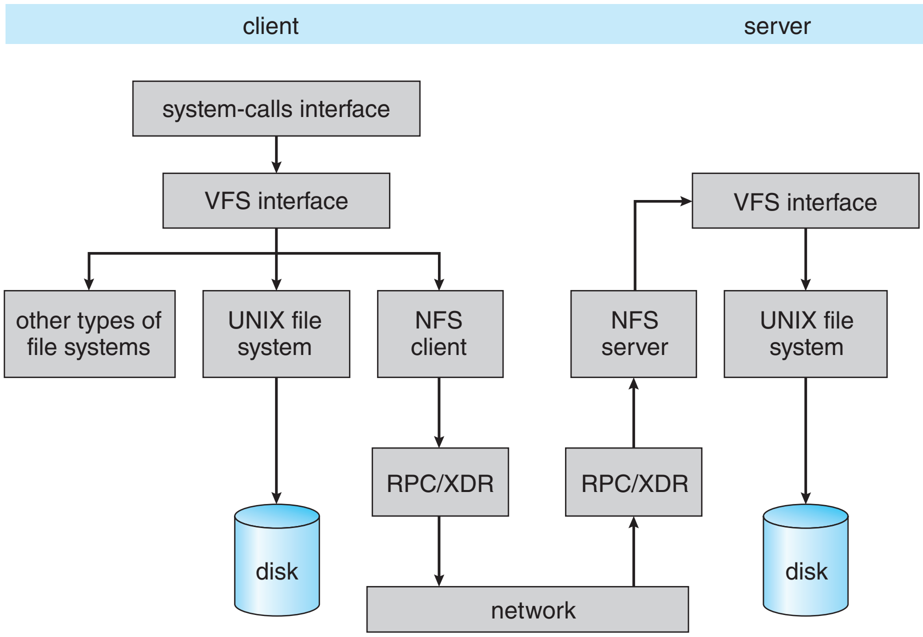


Figure 15.8 Schematic view of the NFS architecture.

NFS is integrated into the operating system via a VFS. As an illustration of the architecture, let's trace how an operation on an already-open remote file is handled (follow the example in Figure 15.8). The client initiates the operation with a regular system call. The operating-system layer maps this call to a VFS operation on the appropriate vnode. The VFS layer identifies the file as a remote one and invokes the appropriate NFS procedure. An RPC call is made to the NFS service layer at the remote server. This call is reinjected to the VFS layer on the remote system, which finds that it is local and invokes the appropriate file-system operation. This path is retraced to return the result. An advantage of this architecture is that the client and the server are identical; thus, a machine may be a client, or a server, or both. The actual service on each server is performed by kernel threads.

15.8.4 Path-Name Translation

Path-name translation in NFS involves the parsing of a path name such as `/usr/local/dir1/file.txt` into separate directory entries, or components: (1) `usr`, (2) `local`, and (3) `dir1`. Path-name translation is done by breaking the path into component names and performing a separate NFS `lookup call` for every pair of component name and directory vnode. Once a mount point is crossed, every component lookup causes a separate RPC to the server. This expensive path-name-traversal scheme is needed, since the layout of each client's logical name space is unique, dictated by the mounts the client has performed. It would be much more efficient to hand a server a path name and receive a target vnode once a mount point is encountered. At any point, however, there might be another mount point for the particular client of which the stateless server is unaware.

So that lookup is fast, a directory-name-lookup cache on the client side holds the vnodes for remote directory names. This cache speeds up references to files with the same initial path name. The directory cache is discarded when attributes returned from the server do not match the attributes of the cached vnode.

Recall that some implementations of NFS allow mounting a remote file system on top of another already-mounted remote file system (a cascading mount). When a client has a cascading mount, more than one server can be involved in a path-name traversal. However, when a client does a lookup on a directory on which the server has mounted a file system, the client sees the underlying directory instead of the mounted directory.

15.8.5 Remote Operations

With the exception of opening and closing files, there is an almost one-to-one correspondence between the regular UNIX system calls for file operations and the NFS protocol RPCs. Thus, a remote file operation can be translated directly to the corresponding RPC. Conceptually, NFS adheres to the remote-service paradigm; but in practice, buffering and caching techniques are employed for the sake of performance. No direct correspondence exists between a remote operation and an RPC. Instead, file blocks and file attributes are fetched by the RPCs and are cached locally. Future remote operations use the cached data, subject to consistency constraints.

There are two caches: the file-attribute (inode-information) cache and the file-blocks cache. When a file is opened, the kernel checks with the remote server to determine whether to fetch or revalidate the cached attributes. The cached file blocks are used only if the corresponding cached attributes are up to date. The attribute cache is updated whenever new attributes arrive from the server. Cached attributes are, by default, discarded after 60 seconds. Both read-ahead and delayed-write techniques are used between the server and the client. Clients do not free delayed-write blocks until the server confirms that the data have been written to disk. Delayed-write is retained even when a file is opened concurrently, in conflicting modes. Hence, UNIX semantics (Section 15.7.1) are not preserved.

Tuning the system for performance makes it difficult to characterize the consistency semantics of NFS. New files created on a machine may not be visible elsewhere for 30 seconds. Furthermore, writes to a file at one site may or may not be visible at other sites that have this file open for reading. New opens of a file observe only the changes that have already been flushed to the server. Thus, NFS provides neither strict emulation of UNIX semantics nor the session semantics of Andrew (Section 15.7.2). In spite of these drawbacks, the utility and good performance of the mechanism make it the most widely used multi-vendor-distributed system in operation.

15.9 Summary

- General-purpose operating systems provide many file-system types, from special-purpose through general.

- Volumes containing file systems can be mounted into the computer's file-system space.
- Depending on the operating system, the file-system space is seamless (mounted file systems integrated into the directory structure) or distinct (each mounted file system having its own designation).
- At least one file system must be bootable for the system to be able to start —that is, it must contain an operating system. The boot loader is run first; it is a simple program that is able to find the kernel in the file system, load it, and start its execution. Systems can contain multiple bootable partitions, letting the administrator choose which to run at boot time.
- Most systems are multi-user and thus must provide a method for file sharing and file protection. Frequently, files and directories include metadata, such as owner, user, and group access permissions.
- Mass storage partitions are used either for raw block I/O or for file systems. Each file system resides in a volume, which can be composed of one partition or multiple partitions working together via a volume manager.
- To simplify implementation of multiple file systems, an operating system can use a layered approach, with a virtual file-system interface making access to possibly dissimilar file systems seamless.
- Remote file systems can be implemented simply by using a program such as `ftp` or the web servers and clients in the World Wide Web, or with more functionality via a client–server model. Mount requests and user IDs must be authenticated to prevent unapproved access.
- Client–server facilities do not natively share information, but a distributed information system such as DNS can be used to allow such sharing, providing a unified user name space, password management, and system identification. For example, Microsoft CIFS uses active directory, which employs a version of the Kerberos network authentication protocol to provide a full set of naming and authentication services among the computers in a network.
- Once file sharing is possible, a consistency semantics model must be chosen and implemented to moderate multiple concurrent access to the same file. Semantics models include UNIX, session, and immutable-shared-files semantics.
- NFS is an example of a remote file system, providing clients with seamless access to directories, files, and even entire file systems. A full-featured remote file system includes a communication protocol with remote operations and path-name translation.

Practice Exercises

15.1 Explain how the VFS layer allows an operating system to support multiple types of file systems easily.

15.2 Why have more than one file system type on a given system?

15.3 On a Unix or Linux system that implements the procfs file system, determine how to use the procfs interface to explore the process name space. What aspects of processes can be viewed via this interface? How would the same information be gathered on a system lacking the procfs file system?

15.4 Why do some systems integrate mounted file systems into the root file system naming structure, while others use a separate naming method for mounted file systems?

15.5 Given a remote file access facility such as `ftp`, why were remote file systems like NFS created?

Further Reading

The internals of the BSD UNIX system are covered in full in [McKusick et al. (2015)]. Details concerning file systems for Linux can be found in [Love (2010)].

The network file system (NFS) is discussed in [Callaghan (2000)]. NFS Version 4 is a standard described at http://www.ietf.org/rfc/rfc3530.txt. [Ousterhout (1991)] discusses the role of distributed state in networked file systems. NFS and the UNIX file system (UFS) are described in [Mauro and McDougall (2007)].

The Kerberos network authentication protocol is explored in https://web.mit.edu/kerberos/.

Bibliography

[Callaghan (2000)] B. Callaghan, *NFS Illustrated*, Addison-Wesley (2000).

[Love (2010)] R. Love, *Linux Kernel Development*, Third Edition, Developer's Library (2010).

[Mauro and McDougall (2007)] J. Mauro and R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall (2007).

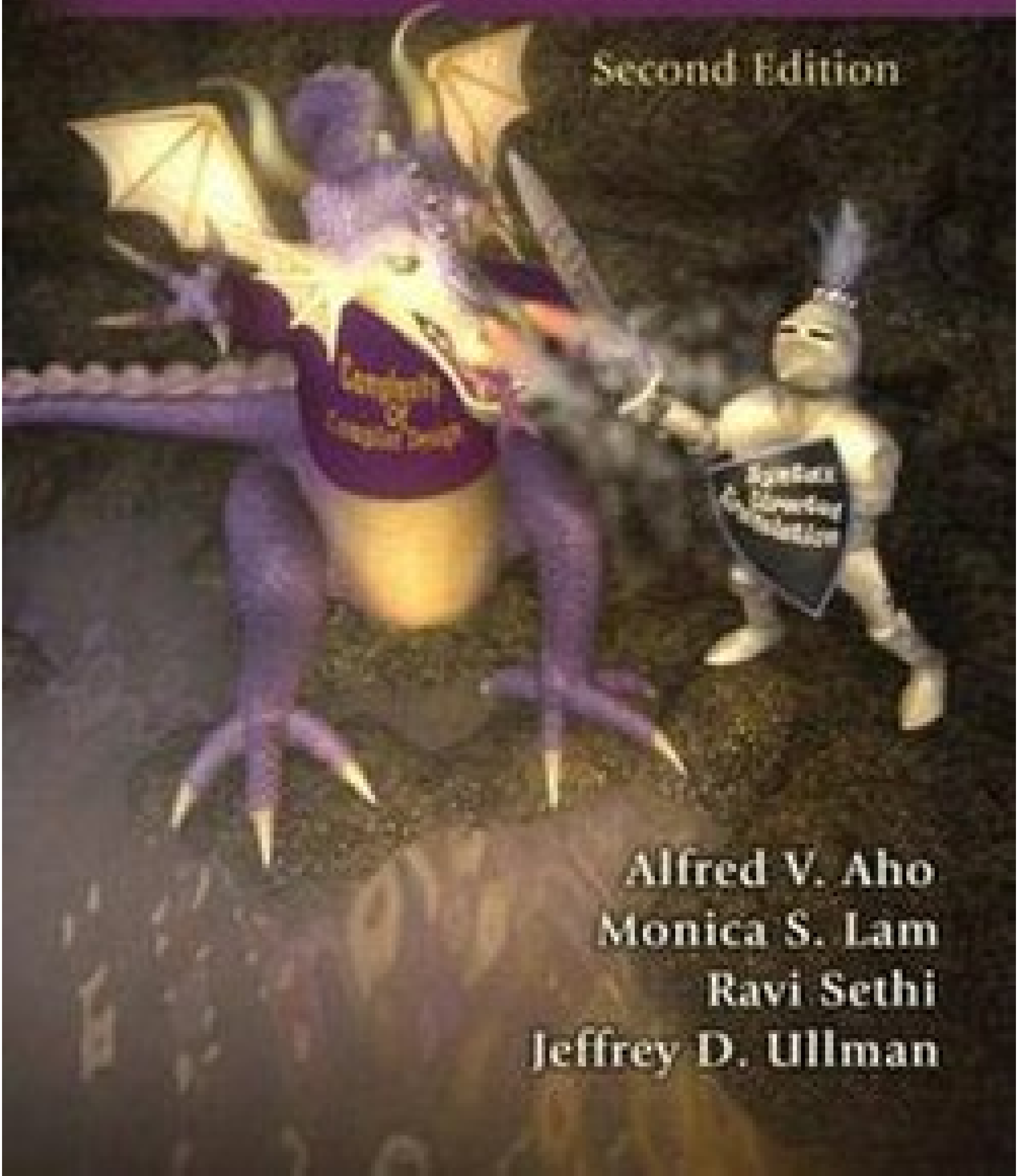
[McKusick et al. (2015)] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD UNIX Operating System–Second Edition*, Pearson (2015).

[Ousterhout (1991)] J. Ousterhout. "The Role of Distributed State". *In CMU Computer Science: a 25th Anniversary Commemorative*, R. F. Rashid, Ed., Addison-Wesley (1991).

Compilers

Principles, Techniques, & Tools

Second Edition



Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman

6.2 Declarations

A complete B-Minor program is a sequence of declarations. Each declaration states the existence of a variable or a function. A variable declaration may optionally give an initializing value. If none is given, it is given a default value of zero. A function declaration may optionally give the body of the function in code; if no body is given, then the declaration serves as a prototype for a function declared elsewhere.

For example, the following are all valid declarations:

```
b: boolean;
s: string = "hello";
f: function integer ( x: integer ) = { return x*x; }
```

A declaration is represented by a `decl` structure that gives the name, type, value (if an expression), code (if a function), and a pointer to the next declaration in the program:

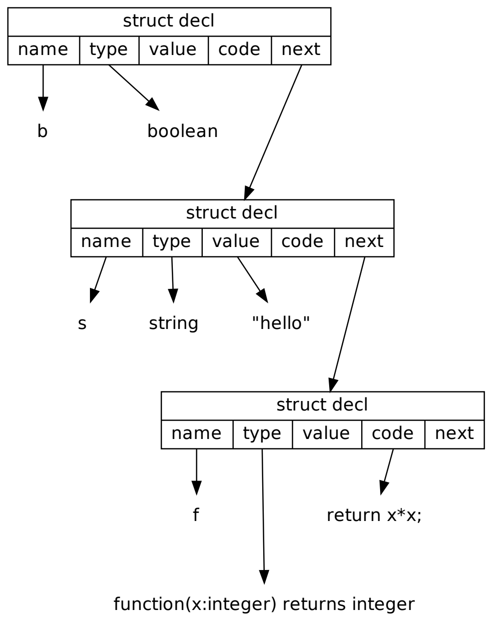
```
struct decl {
    char *name;
    struct type *type;
    struct expr *value;
    struct stmt *code;
    struct decl *next;
};
```

Because we will be creating a lot of these structures, you will need a factory function that allocates a structure and initializes its fields, like this:

```
struct decl * decl_create( char *name,
                           struct type *type,
                           struct expr *value,
                           struct stmt *code,
                           struct decl *next )
{
    struct decl *d = malloc(sizeof(*d));
    d->name = name;
    d->type = type;
    d->value = value;
    d->code = code;
    d->next = next;
    return d;
}
```

(You will need to write similar code for statements, expressions, etc, but we won't keep repeating it here.)

The three declarations on the preceding page can be represented graphically as a linked list, like this:



Note that some of the fields point to nothing: these would be represented by a null pointer, which we omit for clarity. Also, our picture is incomplete and must be expanded: the items representing types, expressions, and statements are all complex structures themselves that we must describe.

6.3 Statements

The body of a function consists of a sequence of statements. A statement indicates that the program is to take a particular action in the order specified, such as computing a value, performing a loop, or choosing between branches of an alternative. A statement can also be a declaration of a local variable. Here is the `stmt` structure:

```
struct stmt {
    stmt_t kind;
    struct decl *decl;
    struct expr *init_expr;
    struct expr *expr;
    struct expr *next_expr;
    struct stmt *body;
    struct stmt *else_body;
    struct stmt *next;
};
```

```
typedef enum {
    STMT_DECL,
    STMT_EXPR,
    STMT_IF_ELSE,
    STMT_FOR,
    STMT_PRINT,
    STMT_RETURN,
    STMT_BLOCK
} stmt_t;
```

The `kind` field indicates what kind of statement it is:

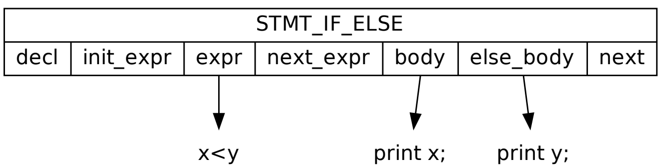
- `STMT_DECL` indicates a (local) declaration, and the `decl` field will point to it.
- `STMT_EXPR` indicates an expression statement and the `expr` field will point to it.
- `STMT_IF_ELSE` indicates an if-else expression such that the `expr` field will point to the control expression, the `body` field to the statements executed if it is true, and the `else_body` field to the statements executed if it is false.
- `STMT_FOR` indicates a for-loop, such that `init_expr`, `expr`, and `next_expr` are the three expressions in the loop header, and `body` points to the statements in the loop.
- `STMT_PRINT` indicates a `print` statement, and `expr` points to the expressions to print.
- `STMT_RETURN` indicates a `return` statement, and `expr` points to the expression to return.
- `STMT_BLOCK` indicates a block of statements inside curly braces, and `body` points to the contained statements.

And, as we did with declarations, we require a function `stmt_create` to create and return a statement structure:

```
struct stmt * stmt_create( stmt_t kind,
    struct decl *decl, struct expr *init_expr,
    struct expr *expr, struct expr *next_expr,
    struct stmt *body, struct stmt *else_body,
    struct stmt *next );
```

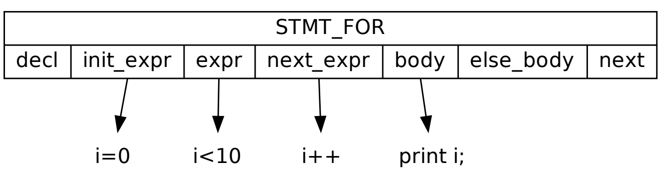
This structure has a lot of fields, but each one serves a purpose and is used when necessary for a particular kind of statement. For example, an if-else statement only uses the `expr`, `body`, and `else_body` fields, leaving the rest null:

```
if( x<y ) print x; else print y;
```



A for-loop uses the three `expr` fields to represent the three parts of the loop control, and the `body` field to represent the code being executed:

```
for(i=0;i<10;i++) print i;
```



6.4 Expressions

Expressions are implemented much like the simple expression AST shown in Chapter 5. The difference is that we need many more binary types: one for every operator in the language, including arithmetic, logical, comparison, assignment, and so forth. We also need one for every type of leaf value, including variable names, constant values, and so forth. The `name` field will be set for `EXPR_NAME`, the `integer_value` field for `EXPR_INTEGER_LITERAL`, and so on. You may need to add values and types to this structure as you expand your compiler.

```
struct expr {
    expr_t kind;
    struct expr *left;
    struct expr *right;

    const char *name;
    int integer_value;
    const char *
       string_literal;
};
```

```
typedef enum {
    EXPR_ADD,
    EXPR_SUB,
    EXPR_MUL,
    EXPR_DIV,
    ...
    EXPR_NAME,
    EXPR_INTEGER_LITERAL,
    EXPR_STRING_LITERAL
} expr_t;
```

As before, you should create a factory for a binary operator:

```
struct expr * expr_create( expr_t kind,
                           struct expr *L,
                           struct expr *R );
```

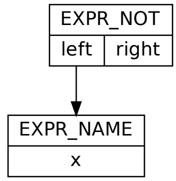
And then a factory for each of the leaf types:

```
struct expr * expr_create_name( const char *name );
struct expr * expr_create_integer_literal( int i );
struct expr * expr_create_boolean_literal( int b );
struct expr * expr_create_char_literal( char c );
struct expr * expr_create_string_literal
                                 ( const char *str );
```

Note that you can store the integer, boolean, and character literal values all in the `integer_value` field.

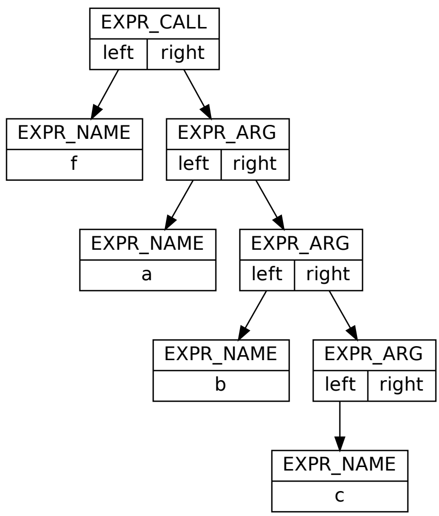
A few cases deserve special mention. Unary operators like logical-not typically have their sole argument in the `left` pointer:

`!x`



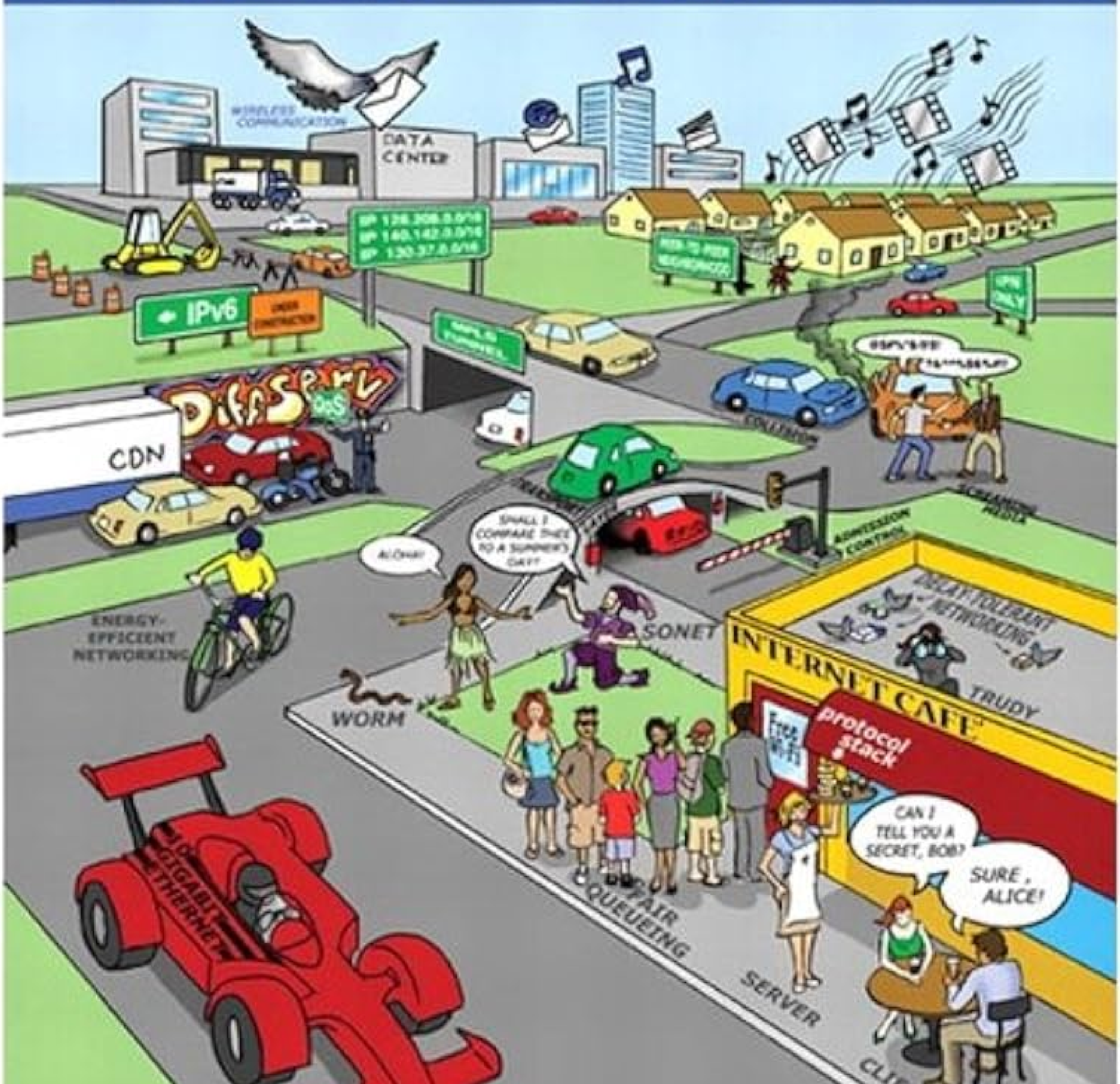
A function call is constructed by creating an `EXPR_CALL` node, such that the left-hand side is the function name, and the right hand side is an unbalanced tree of `EXPR_ARG` nodes. While this looks a bit awkward, it allows us to express a linked list using a tree, and will simplify the handling of function call arguments on the stack during code generation.

`f(a,b,c)`



FIFTH EDITION

COMPUTER NETWORKS



TANENBAUM | WETHERALL

by cameras (that sense light) and displays (that emit light using LEDs and other technology). Data communication can be layered on top of these displays by encoding information in the pattern at which LEDs turn on and off that is below the threshold of human perception. Communicating with visible light in this way is inherently safe and creates a low-speed network in the immediate vicinity of the display. This could enable all sorts of fanciful ubiquitous computing scenarios. The flashing lights on emergency vehicles might alert nearby traffic lights and vehicles to help clear a path. Informational signs might broadcast maps. Even festive lights might broadcast songs that are synchronized with their display.

2.4 COMMUNICATION SATELLITES

In the 1950s and early 1960s, people tried to set up communication systems by bouncing signals off metallized weather balloons. Unfortunately, the received signals were too weak to be of any practical use. Then the U.S. Navy noticed a kind of permanent weather balloon in the sky—the moon—and built an operational system for ship-to-shore communication by bouncing signals off it.

Further progress in the celestial communication field had to wait until the first communication satellite was launched. The key difference between an artificial satellite and a real one is that the artificial one can amplify the signals before sending them back, turning a strange curiosity into a powerful communication system.

Communication satellites have some interesting properties that make them attractive for many applications. In its simplest form, a communication satellite can be thought of as a big microwave repeater in the sky. It contains several **transponders**, each of which listens to some portion of the spectrum, amplifies the incoming signal, and then rebroadcasts it at another frequency to avoid interference with the incoming signal. This mode of operation is known as a **bent pipe**. Digital processing can be added to separately manipulate or redirect data streams in the overall band, or digital information can even be received by the satellite and rebroadcast. Regenerating signals in this way improves performance compared to a bent pipe because the satellite does not amplify noise in the upward signal. The downward beams can be broad, covering a substantial fraction of the earth's surface, or narrow, covering an area only hundreds of kilometers in diameter.

According to Kepler's law, the orbital period of a satellite varies as the radius of the orbit to the 3/2 power. The higher the satellite, the longer the period. Near the surface of the earth, the period is about 90 minutes. Consequently, low-orbit satellites pass out of view fairly quickly, so many of them are needed to provide continuous coverage and ground antennas must track them. At an altitude of about 35,800 km, the period is 24 hours. At an altitude of 384,000 km, the period is about one month, as anyone who has observed the moon regularly can testify.

A satellite's period is important, but it is not the only issue in determining where to place it. Another issue is the presence of the Van Allen belts, layers of highly charged particles trapped by the earth's magnetic field. Any satellite flying within them would be destroyed fairly quickly by the particles. These factors lead to three regions in which satellites can be placed safely. These regions and some of their properties are illustrated in Fig. 2-15. Below we will briefly describe the satellites that inhabit each of these regions.

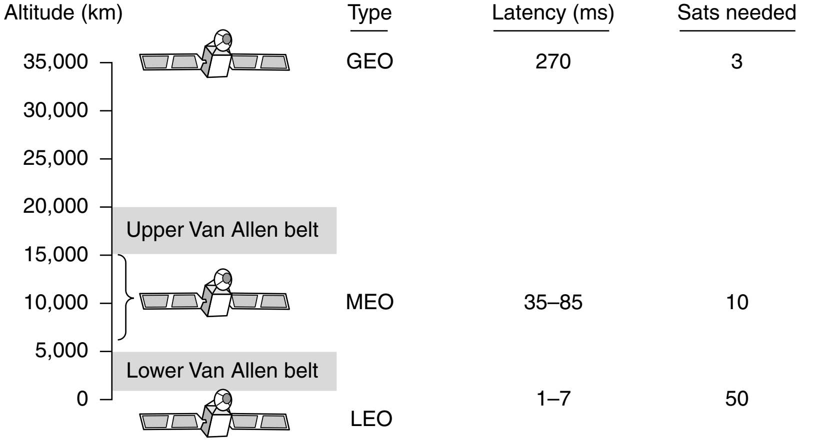


Figure 2-15. Communication satellites and some of their properties, including altitude above the earth, round-trip delay time, and number of satellites needed for global coverage.

2.4.1 Geostationary Satellites

In 1945, the science fiction writer Arthur C. Clarke calculated that a satellite at an altitude of 35,800 km in a circular equatorial orbit would appear to remain motionless in the sky, so it would not need to be tracked (Clarke, 1945). He went on to describe a complete communication system that used these (manned) **geostationary satellites**, including the orbits, solar panels, radio frequencies, and launch procedures. Unfortunately, he concluded that satellites were impractical due to the impossibility of putting power-hungry, fragile vacuum tube amplifiers into orbit, so he never pursued this idea further, although he wrote some science fiction stories about it.

The invention of the transistor changed all that, and the first artificial communication satellite, Telstar, was launched in July 1962. Since then, communication satellites have become a multibillion dollar business and the only aspect of outer space that has become highly profitable. These high-flying satellites are often called **GEO** (**Geostationary Earth Orbit**) satellites.

With current technology, it is unwise to have geostationary satellites spaced much closer than 2 degrees in the 360-degree equatorial plane, to avoid interference. With a spacing of 2 degrees, there can only be 360/2 = 180 of these satellites in the sky at once. However, each transponder can use multiple frequencies and polarizations to increase the available bandwidth.

To prevent total chaos in the sky, orbit slot allocation is done by ITU. This process is highly political, with countries barely out of the stone age demanding "their" orbit slots (for the purpose of leasing them to the highest bidder). Other countries, however, maintain that national property rights do not extend up to the moon and that no country has a legal right to the orbit slots above its territory. To add to the fight, commercial telecommunication is not the only application. Television broadcasters, governments, and the military also want a piece of the orbiting pie.

Modern satellites can be quite large, weighing over 5000 kg and consuming several kilowatts of electric power produced by the solar panels. The effects of solar, lunar, and planetary gravity tend to move them away from their assigned orbit slots and orientations, an effect countered by on-board rocket motors. This fine-tuning activity is called **station keeping**. However, when the fuel for the motors has been exhausted (typically after about 10 years) the satellite drifts and tumbles helplessly, so it has to be turned off. Eventually, the orbit decays and the satellite reenters the atmosphere and burns up (or very rarely crashes to earth).

Orbit slots are not the only bone of contention. Frequencies are an issue, too, because the downlink transmissions interfere with existing microwave users. Consequently, ITU has allocated certain frequency bands to satellite users. The main ones are listed in Fig. 2-16. The C band was the first to be designated for commercial satellite traffic. Two frequency ranges are assigned in it, the lower one for downlink traffic (from the satellite) and the upper one for uplink traffic (to the satellite). To allow traffic to go both ways at the same time, two channels are required. These channels are already overcrowded because they are also used by the common carriers for terrestrial microwave links. The L and S bands were added by international agreement in 2000. However, they are narrow and also crowded.

Band	Downlink	Uplink	Bandwidth	Problems
L	1.5 GHz	1.6 GHz	15 MHz	Low bandwidth; crowded
S	1.9 GHz	2.2 GHz	70 MHz	Low bandwidth; crowded
C	4.0 GHz	6.0 GHz	500 MHz	Terrestrial interference
Ku	11 GHz	14 GHz	500 MHz	Rain
Ka	20 GHz	30 GHz	3500 MHz	Rain, equipment cost

Figure 2-16. The principal satellite bands.

The next-highest band available to commercial telecommunication carriers is the Ku (K under) band. This band is not (yet) congested, and at its higher frequencies, satellites can be spaced as close as 1 degree. However, another problem exists: rain. Water absorbs these short microwaves well. Fortunately, heavy storms are usually localized, so using several widely separated ground stations instead of just one circumvents the problem, but at the price of extra antennas, extra cables, and extra electronics to enable rapid switching between stations. Bandwidth has also been allocated in the Ka (K above) band for commercial satellite traffic, but the equipment needed to use it is expensive. In addition to these commercial bands, many government and military bands also exist.

A modern satellite has around 40 transponders, most often with a 36-MHz bandwidth. Usually, each transponder operates as a bent pipe, but recent satellites have some on-board processing capacity, allowing more sophisticated operation. In the earliest satellites, the division of the transponders into channels was static: the bandwidth was simply split up into fixed frequency bands. Nowadays, each transponder beam is divided into time slots, with various users taking turns. We will study these two techniques (frequency division multiplexing and time division multiplexing) in detail later in this chapter.

The first geostationary satellites had a single spatial beam that illuminated about 1/3 of the earth's surface, called its **footprint**. With the enormous decline in the price, size, and power requirements of microelectronics, a much more sophisticated broadcasting strategy has become possible. Each satellite is equipped with multiple antennas and multiple transponders. Each downward beam can be focused on a small geographical area, so multiple upward and downward transmissions can take place simultaneously. Typically, these so-called **spot beams** are elliptically shaped, and can be as small as a few hundred km in diameter. A communication satellite for the United States typically has one wide beam for the contiguous 48 states, plus spot beams for Alaska and Hawaii.

A recent development in the communication satellite world is the development of low-cost microstations, sometimes called **VSATs** (**Very Small Aperture Terminals**) (Abramson, 2000). These tiny terminals have 1-meter or smaller antennas (versus 10 m for a standard GEO antenna) and can put out about 1 watt of power. The uplink is generally good for up to 1 Mbps, but the downlink is often up to several megabits/sec. Direct broadcast satellite television uses this technology for one-way transmission.

In many VSAT systems, the microstations do not have enough power to communicate directly with one another (via the satellite, of course). Instead, a special ground station, the **hub**, with a large, high-gain antenna is needed to relay traffic between VSATs, as shown in Fig. 2-17. In this mode of operation, either the sender or the receiver has a large antenna and a powerful amplifier. The trade-off is a longer delay in return for having cheaper end-user stations.

VSATs have great potential in rural areas. It is not widely appreciated, but over half the world's population lives more than hour's walk from the nearest

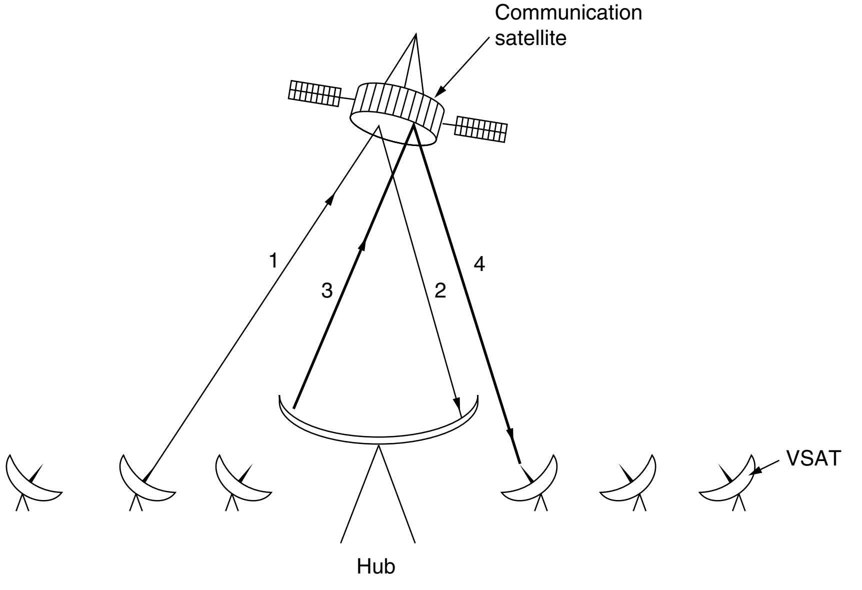


Figure 2-17. VSATs using a hub.

telephone. Stringing telephone wires to thousands of small villages is far beyond the budgets of most Third World governments, but installing 1-meter VSAT dishes powered by solar cells is often feasible. VSATs provide the technology that will wire the world.

Communication satellites have several properties that are radically different from terrestrial point-to-point links. To begin with, even though signals to and from a satellite travel at the speed of light (nearly 300,000 km/sec), the long round-trip distance introduces a substantial delay for GEO satellites. Depending on the distance between the user and the ground station and the elevation of the satellite above the horizon, the end-to-end transit time is between 250 and 300 msec. A typical value is 270 msec (540 msec for a VSAT system with a hub).

For comparison purposes, terrestrial microwave links have a propagation delay of roughly 3 μsec/km, and coaxial cable or fiber optic links have a delay of approximately 5 μsec/km. The latter are slower than the former because electromagnetic signals travel faster in air than in solid materials.

Another important property of satellites is that they are inherently broadcast media. It does not cost more to send a message to thousands of stations within a transponder's footprint than it does to send to one. For some applications, this property is very useful. For example, one could imagine a satellite broadcasting popular Web pages to the caches of a large number of computers spread over a wide area. Even when broadcasting can be simulated with point-to-point lines, satellite broadcasting may be much cheaper. On the other hand, from a privacy point of view, satellites are a complete disaster: everybody can hear everything. Encryption is essential when security is required.

Satellites also have the property that the cost of transmitting a message is independent of the distance traversed. A call across the ocean costs no more to service than a call across the street. Satellites also have excellent error rates and can be deployed almost instantly, a major consideration for disaster response and military communication.

2.4.2 Medium-Earth Orbit Satellites

At much lower altitudes, between the two Van Allen belts, we find the **MEO** (**Medium-Earth Orbit**) satellites. As viewed from the earth, these drift slowly in longitude, taking something like 6 hours to circle the earth. Accordingly, they must be tracked as they move through the sky. Because they are lower than the GEOs, they have a smaller footprint on the ground and require less powerful transmitters to reach them. Currently they are used for navigation systems rather than telecommunications, so we will not examine them further here. The constellation of roughly 30 **GPS** (**Global Positioning System**) satellites orbiting at about 20,200 km are examples of MEO satellites.

2.4.3 Low-Earth Orbit Satellites

Moving down in altitude, we come to the **LEO** (**Low-Earth Orbit**) satellites. Due to their rapid motion, large numbers of them are needed for a complete system. On the other hand, because the satellites are so close to the earth, the ground stations do not need much power, and the round-trip delay is only a few milliseconds. The launch cost is substantially cheaper too. In this section we will examine two examples of satellite constellations for voice service, Iridium and Globalstar.

For the first 30 years of the satellite era, low-orbit satellites were rarely used because they zip into and out of view so quickly. In 1990, Motorola broke new ground by filing an application with the FCC asking for permission to launch 77 low-orbit satellites for the **Iridium** project (element 77 is iridium). The plan was later revised to use only 66 satellites, so the project should have been renamed Dysprosium (element 66), but that probably sounded too much like a disease. The idea was that as soon as one satellite went out of view, another would replace it. This proposal set off a feeding frenzy among other communication companies. All of a sudden, everyone wanted to launch a chain of low-orbit satellites.

After seven years of cobbling together partners and financing, communication service began in November 1998. Unfortunately, the commercial demand for large, heavy satellite telephones was negligible because the mobile phone network had grown in a spectacular way since 1990. As a consequence, Iridium was not profitable and was forced into bankruptcy in August 1999 in one of the most spectacular corporate fiascos in history. The satellites and other assets (worth $5 billion) were later purchased by an investor for $25 million at a kind of extraterrestrial garage sale. Other satellite business ventures promptly followed suit.

The Iridium service restarted in March 2001 and has been growing ever since. It provides voice, data, paging, fax, and navigation service everywhere on land, air, and sea, via hand-held devices that communicate directly with the Iridium satellites. Customers include the maritime, aviation, and oil exploration industries, as well as people traveling in parts of the world lacking a telecom infrastructure (e.g., deserts, mountains, the South Pole, and some Third World countries).

The Iridium satellites are positioned at an altitude of 750 km, in circular polar orbits. They are arranged in north-south necklaces, with one satellite every 32 degrees of latitude, as shown in Fig. 2-18. Each satellite has a maximum of 48 cells (spot beams) and a capacity of 3840 channels, some of which are used for paging and navigation, while others are used for data and voice.

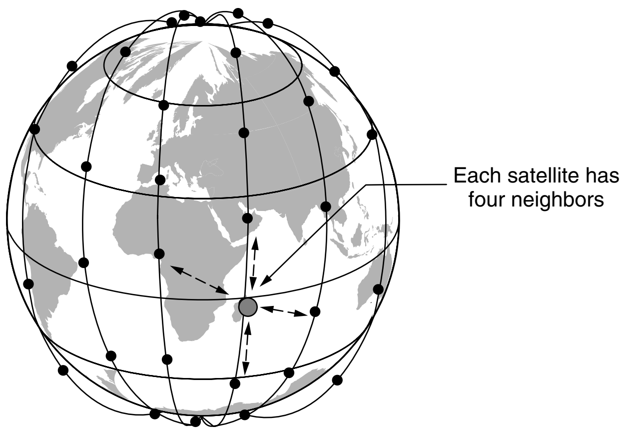


Figure 2-18. The Iridium satellites form six necklaces around the earth.

With six satellite necklaces the entire earth is covered, as suggested by Fig. 2-18. An interesting property of Iridium is that communication between distant customers takes place in space, as shown in Fig. 2-19(a). Here we see a caller at the North Pole contacting a satellite directly overhead. Each satellite has four neighbors with which it can communicate, two in the same necklace (shown) and two in adjacent necklaces (not shown). The satellites relay the call across this grid until it is finally sent down to the callee at the South Pole.

An alternative design to Iridium is **Globalstar**. It is based on 48 LEO satellites but uses a different switching scheme than that of Iridium. Whereas Iridium relays calls from satellite to satellite, which requires sophisticated switching equipment in the satellites, Globalstar uses a traditional bent-pipe design. The call originating at the North Pole in Fig. 2-19(b) is sent back to earth and picked

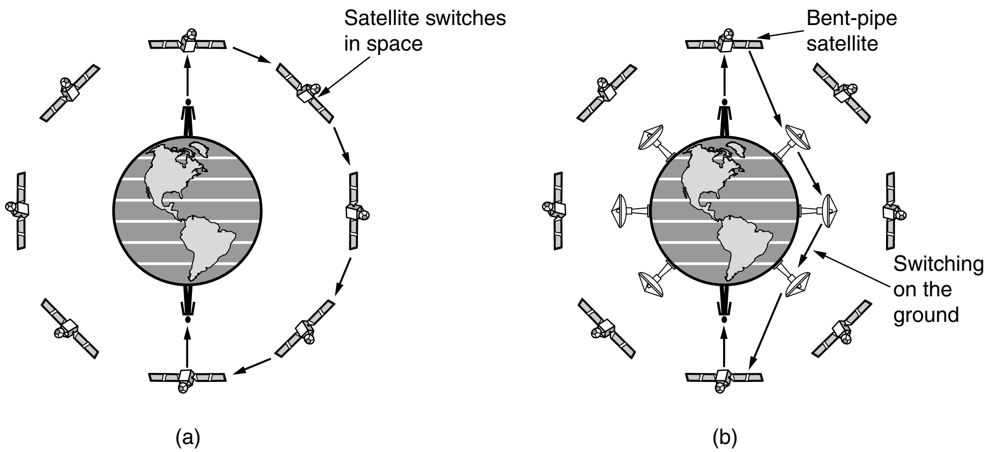


Figure 2-19. (a) Relaying in space. (b) Relaying on the ground.

up by the large ground station at Santa's Workshop. The call is then routed via a terrestrial network to the ground station nearest the callee and delivered by a bent-pipe connection as shown. The advantage of this scheme is that it puts much of the complexity on the ground, where it is easier to manage. Also, the use of large ground station antennas that can put out a powerful signal and receive a weak one means that lower-powered telephones can be used. After all, the telephone puts out only a few milliwatts of power, so the signal that gets back to the ground station is fairly weak, even after having been amplified by the satellite.

Satellites continue to be launched at a rate of around 20 per year, including ever-larger satellites that now weigh over 5000 kilograms. But there are also very small satellites for the more budget-conscious organization. To make space research more accessible, academics from Cal Poly and Stanford got together in 1999 to define a standard for miniature satellites and an associated launcher that would greatly lower launch costs (Nugent et al., 2008). **CubeSats** are satellites in units of 10 cm × 10 cm × 10 cm cubes, each weighing no more than 1 kilogram, that can be launched for as little as $40,000 each. The launcher flies as a secondary payload on commercial space missions. It is basically a tube that takes up to three units of cubesats and uses springs to release them into orbit. Roughly 20 cubesats have launched so far, with many more in the works. Most of them communicate with ground stations on the UHF and VHF bands.

2.4.4 Satellites Versus Fiber

A comparison between satellite communication and terrestrial communication is instructive. As recently as 25 years ago, a case could be made that the future of communication lay with communication satellites. After all, the telephone system had changed little in the previous 100 years and showed no signs of changing in the next 100 years. This glacial movement was caused in no small part by the regulatory environment in which the telephone companies were expected to provide good voice service at reasonable prices (which they did), and in return got a guaranteed profit on their investment. For people with data to transmit, 1200-bps modems were available. That was pretty much all there was.

The introduction of competition in 1984 in the United States and somewhat later in Europe changed all that radically. Telephone companies began replacing their long-haul networks with fiber and introduced high-bandwidth services like ADSL (Asymmetric Digital Subscriber Line). They also stopped their long-time practice of charging artificially high prices to long-distance users to subsidize local service. All of a sudden, terrestrial fiber connections looked like the winner.

Nevertheless, communication satellites have some major niche markets that fiber does not (and, sometimes, cannot) address. First, when rapid deployment is critical, satellites win easily. A quick response is useful for military communication systems in times of war and disaster response in times of peace. Following the massive December 2004 Sumatra earthquake and subsequent tsunami, for example, communications satellites were able to restore communications to first responders within 24 hours. This rapid response was possible because there is a developed satellite service provider market in which large players, such as Intelsat with over 50 satellites, can rent out capacity pretty much anywhere it is needed. For customers served by existing satellite networks, a VSAT can be set up easily and quickly to provide a megabit/sec link to elsewhere in the world.

A second niche is for communication in places where the terrestrial infrastructure is poorly developed. Many people nowadays want to communicate everywhere they go. Mobile phone networks cover those locations with good population density, but do not do an adequate job in other places (e.g., at sea or in the desert). Conversely, Iridium provides voice service everywhere on Earth, even at the South Pole. Terrestrial infrastructure can also be expensive to install, depending on the terrain and necessary rights of way. Indonesia, for example, has its own satellite for domestic telephone traffic. Launching one satellite was cheaper than stringing thousands of undersea cables among the 13,677 islands in the archipelago.

A third niche is when broadcasting is essential. A message sent by satellite can be received by thousands of ground stations at once. Satellites are used to distribute much network TV programming to local stations for this reason. There is now a large market for satellite broadcasts of digital TV and radio directly to end users with satellite receivers in their homes and cars. All sorts of other content can be broadcast too. For example, an organization transmitting a stream of stock, bond, or commodity prices to thousands of dealers might find a satellite system to be much cheaper than simulating broadcasting on the ground.

In short, it looks like the mainstream communication of the future will be terrestrial fiber optics combined with cellular radio, but for some specialized uses, satellites are better. However, there is one caveat that applies to all of this: economics. Although fiber offers more bandwidth, it is conceivable that terrestrial and satellite communication could compete aggressively on price. If advances in technology radically cut the cost of deploying a satellite (e.g., if some future space vehicle can toss out dozens of satellites on one launch) or low-orbit satellites catch on in a big way, it is not certain that fiber will win all markets.

2.5 DIGITAL MODULATION AND MULTIPLEXING

Now that we have studied the properties of wired and wireless channels, we turn our attention to the problem of sending digital information. Wires and wireless channels carry analog signals such as continuously varying voltage, light intensity, or sound intensity. To send digital information, we must devise analog signals to represent bits. The process of converting between bits and signals that represent them is called **digital modulation**.

We will start with schemes that directly convert bits into a signal. These schemes result in **baseband transmission**, in which the signal occupies frequencies from zero up to a maximum that depends on the signaling rate. It is common for wires. Then we will consider schemes that regulate the amplitude, phase, or frequency of a carrier signal to convey bits. These schemes result in **passband transmission**, in which the signal occupies a band of frequencies around the frequency of the carrier signal. It is common for wireless and optical channels for which the signals must reside in a given frequency band.

Channels are often shared by multiple signals. After all, it is much more convenient to use a single wire to carry several signals than to install a wire for every signal. This kind of sharing is called **multiplexing**. It can be accomplished in several different ways. We will present methods for time, frequency, and code division multiplexing.

The modulation and multiplexing techniques we describe in this section are all widely used for wires, fiber, terrestrial wireless, and satellite channels. In the following sections, we will look at examples of networks to see them in action.

2.5.1 Baseband Transmission

The most straightforward form of digital modulation is to use a positive voltage to represent a 1 and a negative voltage to represent a 0. For an optical fiber, the presence of light might represent a 1 and the absence of light might represent a 0. This scheme is called **NRZ** (**Non-Return-to-Zero**). The odd name is for historical reasons, and simply means that the signal follows the data. An example is shown in Fig. 2-20(b).

Once sent, the NRZ signal propagates down the wire. At the other end, the receiver converts it into bits by sampling the signal at regular intervals of time.

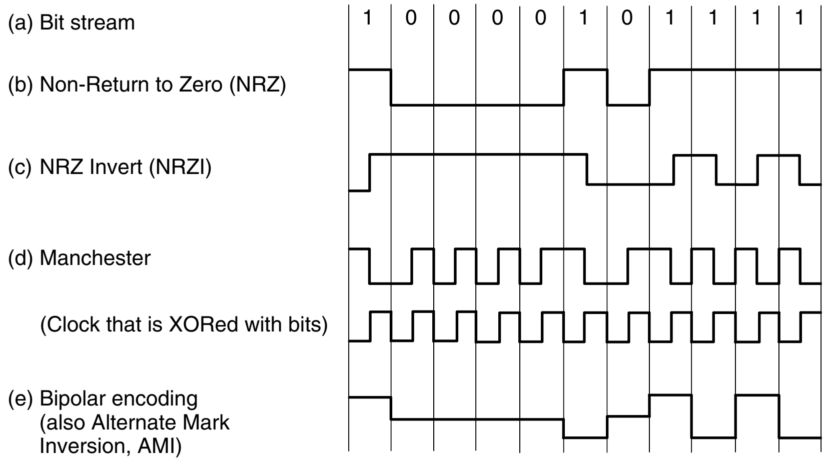


Figure 2-20. Line codes: (a) Bits, (b) NRZ, (c) NRZI, (d) Manchester, (e) Bipolar or AMI.

This signal will not look exactly like the signal that was sent. It will be attenuated and distorted by the channel and noise at the receiver. To decode the bits, the receiver maps the signal samples to the closest symbols. For NRZ, a positive voltage will be taken to indicate that a 1 was sent and a negative voltage will be taken to indicate that a 0 was sent.

NRZ is a good starting point for our studies because it is simple, but it is seldom used by itself in practice. More complex schemes can convert bits to signals that better meet engineering considerations. These schemes are called **line codes**. Below, we describe line codes that help with bandwidth efficiency, clock recovery, and DC balance.

Bandwidth Efficiency

With NRZ, the signal may cycle between the positive and negative levels up to every 2 bits (in the case of alternating 1s and 0s). This means that we need a bandwidth of at least $B/2$ Hz when the bit rate is B bits/sec. This relation comes from the Nyquist rate [Eq. (2-2)]. It is a fundamental limit, so we cannot run NRZ faster without using more bandwidth. Bandwidth is often a limited resource, even for wired channels, Higher-frequency signals are increasingly attenuated, making them less useful, and higher-frequency signals also require faster electronics.

One strategy for using limited bandwidth more efficiently is to use more than two signaling levels. By using four voltages, for instance, we can send 2 bits at once as a single **symbol**. This design will work as long as the signal at the receiver is sufficiently strong to distinguish the four levels. The rate at which the signal changes is then half the bit rate, so the needed bandwidth has been reduced.

We call the rate at which the signal changes the **symbol rate** to distinguish it from the **bit rate**. The bit rate is the symbol rate multiplied by the number of bits per symbol. An older name for the symbol rate, particularly in the context of devices called telephone modems that convey digital data over telephone lines, is the **baud rate**. In the literature, the terms "bit rate" and "baud rate" are often used incorrectly.

Note that the number of signal levels does not need to be a power of two. Often it is not, with some of the levels used for protecting against errors and simplifying the design of the receiver.

Clock Recovery

For all schemes that encode bits into symbols, the receiver must know when one symbol ends and the next symbol begins to correctly decode the bits. With NRZ, in which the symbols are simply voltage levels, a long run of 0s or 1s leaves the signal unchanged. After a while it is hard to tell the bits apart, as 15 zeros look much like 16 zeros unless you have a very accurate clock.

Accurate clocks would help with this problem, but they are an expensive solution for commodity equipment. Remember, we are timing bits on links that run at many megabits/sec, so the clock would have to drift less than a fraction of a microsecond over the longest permitted run. This might be reasonable for slow links or short messages, but it is not a general solution.

One strategy is to send a separate clock signal to the receiver. Another clock line is no big deal for computer buses or short cables in which there are many lines in parallel, but it is wasteful for most network links since if we had another line to send a signal we could use it to send data. A clever trick here is to mix the clock signal with the data signal by XORing them together so that no extra line is needed. The results are shown in Fig. 2-20(d). The clock makes a clock transition in every bit time, so it runs at twice the bit rate. When it is XORed with the 0 level it makes a low-to-high transition that is simply the clock. This transition is a logical 0. When it is XORed with the 1 level it is inverted and makes a high-to-low transition. This transition is a logical 1. This scheme is called **Manchester** encoding and was used for classic Ethernet.

The downside of Manchester encoding is that it requires twice as much bandwidth as NRZ because of the clock, and we have learned that bandwidth often matters. A different strategy is based on the idea that we should code the data to ensure that there are enough transitions in the signal. Consider that NRZ will have clock recovery problems only for long runs of 0s and 1s. If there are frequent transitions, it will be easy for the receiver to stay synchronized with the incoming stream of symbols.

As a step in the right direction, we can simplify the situation by coding a 1 as a transition and a 0 as no transition, or vice versa. This coding is called **NRZI** (**Non-Return-to-Zero Inverted**), a twist on NRZ. An example is shown in

Fig. 2-20(c). The popular **USB** (**Universal Serial Bus**) standard for connecting computer peripherals uses NRZI. With it, long runs of 1s do not cause a problem.

Of course, long runs of 0s still cause a problem that we must fix. If we were the telephone company, we might simply require that the sender not transmit too many 0s. Older digital telephone lines in the U.S., called **T1 lines**, did in fact require that no more than 15 consecutive 0s be sent for them to work correctly. To really fix the problem we can break up runs of 0s by mapping small groups of bits to be transmitted so that groups with successive 0s are mapped to slightly longer patterns that do not have too many consecutive 0s.

A well-known code to do this is called **4B/5B**. Every 4 bits is mapped into a5-bit pattern with a fixed translation table. The five bit patterns are chosen so that there will never be a run of more than three consecutive 0s. The mapping is shown in Fig. 2-21. This scheme adds 25% overhead, which is better than the 100% overhead of Manchester encoding. Since there are 16 input combinations and 32 output combinations, some of the output combinations are not used. Putting aside the combinations with too many successive 0s, there are still some codes left. As a bonus, we can use these nondata codes to represent physical layer control signals. For example, in some uses "11111" represents an idle line and "11000" represents the start of a frame.

Data (4B)	Codeword (5B)	Data (4B)	Codeword (5B)
0000	11110	1000	10010
0001	01001	1001	10011
0010	10100	1010	10110
0011	10101	1011	10111
0100	01010	1100	11010
0101	01011	1101	11011
0110	01110	1110	11100
0111	01111	1111	11101

Figure 2-21. 4B/5B mapping.

An alternative approach is to make the data look random, known as scrambling. In this case it is very likely that there will be frequent transitions. A **scrambler** works by XORing the data with a pseudorandom sequence before it is transmitted. This mixing will make the data as random as the pseudorandom sequence (assuming it is independent of the pseudorandom sequence). The receiver then XORs the incoming bits with the same pseudorandom sequence to recover the real data. For this to be practical, the pseudorandom sequence must be easy to create. It is commonly given as the seed to a simple random number generator.

Scrambling is attractive because it adds no bandwidth or time overhead. In fact, it often helps to condition the signal so that it does not have its energy in dominant frequency components (caused by repetitive data patterns) that might radiate electromagnetic interference. Scrambling helps because random signals tend to be "white," or have energy spread across the frequency components.

However, scrambling does not guarantee that there will be no long runs. It is possible to get unlucky occasionally. If the data are the same as the pseudorandom sequence, they will XOR to all 0s. This outcome does not generally occur with a long pseudorandom sequence that is difficult to predict. However, with a short or predictable sequence, it might be possible for malicious users to send bit patterns that cause long runs of 0s after scrambling and cause links to fail. Early versions of the standards for sending IP packets over SONET links in the telephone system had this defect (Malis and Simpson, 1999). It was possible for users to send certain "killer packets" that were guaranteed to cause problems.

Balanced Signals

Signals that have as much positive voltage as negative voltage even over short periods of time are called **balanced signals**. They average to zero, which means that they have no DC electrical component. The lack of a DC component is an advantage because some channels, such as coaxial cable or lines with transformers, strongly attenuate a DC component due to their physical properties. Also, one method of connecting the receiver to the channel called **capacitive coupling** passes only the AC portion of a signal. In either case, if we send a signal whose average is not zero, we waste energy as the DC component will be filtered out.

Balancing helps to provide transitions for clock recovery since there is a mix of positive and negative voltages. It also provides a simple way to calibrate receivers because the average of the signal can be measured and used as a decision threshold to decode symbols. With unbalanced signals, the average may be drift away from the true decision level due to a density of 1s, for example, which would cause more symbols to be decoded with errors.

A straightforward way to construct a balanced code is to use two voltage levels to represent a logical 1, (say +1 V or −1 V) with 0 V representing a logical zero. To send a 1, the transmitter alternates between the +1 V and −1 V levels so that they always average out. This scheme is called **bipolar encoding**. In telephone networks it is called **AMI** (**Alternate Mark Inversion**), building on old terminology in which a 1 is called a "mark" and a 0 is called a "space." An example is given in Fig. 2-20(e).

Bipolar encoding adds a voltage level to achieve balance. Alternatively we can use a mapping like 4B/5B to achieve balance (as well as transitions for clock recovery). An example of this kind of balanced code is the **8B/10B** line code. It maps 8 bits of input to 10 bits of output, so it is 80% efficient, just like the 4B/5B line code. The 8 bits are split into a group of 5 bits, which is mapped to 6 bits, and a group of 3 bits, which is mapped to 4 bits. The 6-bit and 4-bit symbols are then concatenated. In each group, some input patterns can be mapped to balanced output patterns that have the same number of 0s and 1s. For example, "001" is mapped to "1001," which is balanced. But there are not enough combinations for all output patterns to be balanced. For these cases, each input pattern is mapped to two output patterns. One will have an extra 1 and the alternate will have an extra 0. For example, "000" is mapped to both "1011" and its complement "0100." As input bits are mapped to output bits, the encoder remembers the **disparity** from the previous symbol. The disparity is the total number of 0s or 1s by which the signal is out of balance. The encoder then selects either an output pattern or its alternate to reduce the disparity. With 8B/10B, the disparity will be at most 2 bits. Thus, the signal will never be far from balanced. There will also never be more than five consecutive 1s or 0s, to help with clock recovery.

2.5.2 Passband Transmission

Often, we want to use a range of frequencies that does not start at zero to send information across a channel. For wireless channels, it is not practical to send very low frequency signals because the size of the antenna needs to be a fraction of the signal wavelength, which becomes large. In any case, regulatory constraints and the need to avoid interference usually dictate the choice of frequencies. Even for wires, placing a signal in a given frequency band is useful to let different kinds of signals coexist on the channel. This kind of transmission is called passband transmission because an arbitrary band of frequencies is used to pass the signal.

Fortunately, our fundamental results from earlier in the chapter are all in terms of bandwidth, or the width of the frequency band. The absolute frequency values do not matter for capacity. This means that we can take a **baseband** signal that occupies 0 to B Hz and shift it up to occupy a **passband** of S to $S+B$ Hz without changing the amount of information that it can carry, even though the signal will look different. To process a signal at the receiver, we can shift it back down to baseband, where it is more convenient to detect symbols.

Digital modulation is accomplished with passband transmission by regulating or modulating a carrier signal that sits in the passband. We can modulate the amplitude, frequency, or phase of the carrier signal. Each of these methods has a corresponding name. In **ASK** (**Amplitude Shift Keying**), two different amplitudes are used to represent 0 and 1. An example with a nonzero and a zero level is shown in Fig. 2-22(b). More than two levels can be used to represent more symbols. Similarly, with **FSK** (**Frequency Shift Keying**), two or more different tones are used. The example in Fig. 2-21(c) uses just two frequencies. In the simplest form of **PSK** (**Phase Shift Keying**), the carrier wave is systematically shifted 0 or 180 degrees at each symbol period. Because there are two phases, it is called **BPSK** (**Binary Phase Shift Keying**). "Binary" here refers to the two symbols, not that the symbols represent 2 bits. An example is shown in Fig. 2-22(c). A better scheme that uses the channel bandwidth more efficiently is to use four shifts, e.g., 45, 135, 225, or 315 degrees, to transmit 2 bits of information per symbol. This version is called **QPSK** (**Quadrature Phase Shift Keying**).

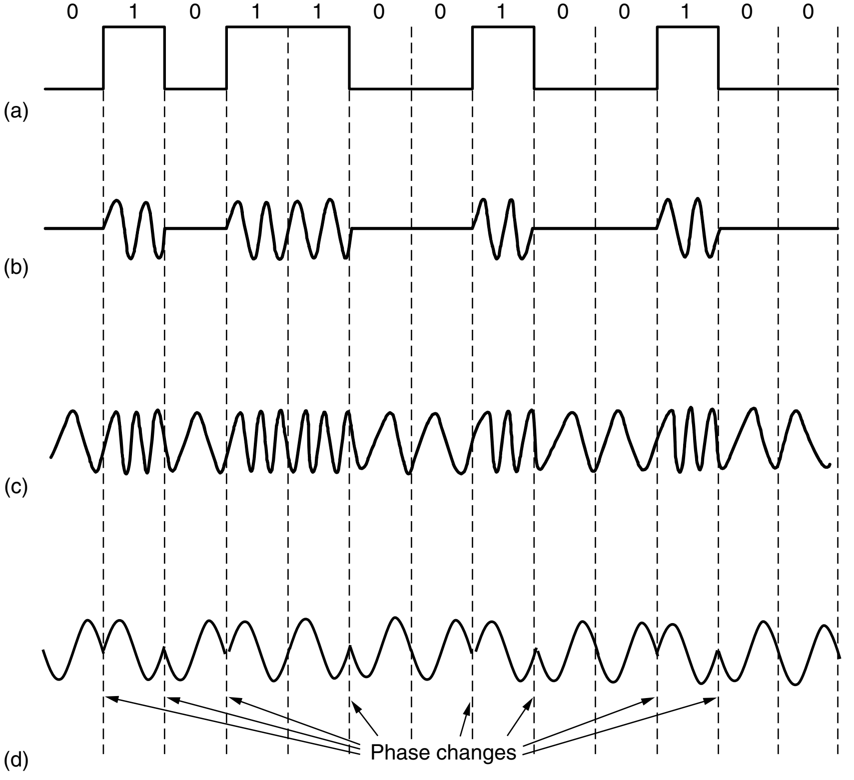


Figure 2-22. (a) A binary signal. (b) Amplitude shift keying. (c) Frequency shift keying. (d) Phase shift keying.

We can combine these schemes and use more levels to transmit more bits per symbol. Only one of frequency and phase can be modulated at a time because they are related, with frequency being the rate of change of phase over time. Usually, amplitude and phase are modulated in combination. Three examples are shown in Fig. 2-23. In each example, the points give the legal amplitude and phase combinations of each symbol. In Fig. 2-23(a), we see equidistant dots at 45, 135, 225, and 315 degrees. The phase of a dot is indicated by the angle a line from it to the origin makes with the positive x-axis. The amplitude of a dot is the distance from the origin. This figure is a representation of QPSK.

This kind of diagram is called a **constellation diagram**. In Fig. 2-23(b) we see a modulation scheme with a denser constellation. Sixteen combinations of amplitudes and phase are used, so the modulation scheme can be used to transmit

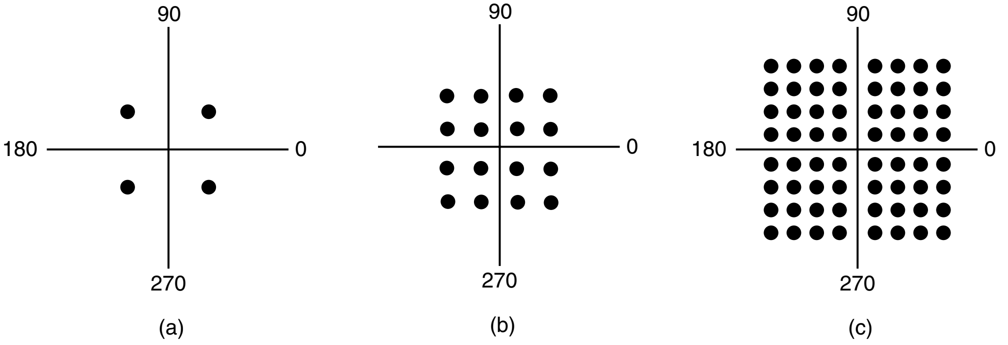


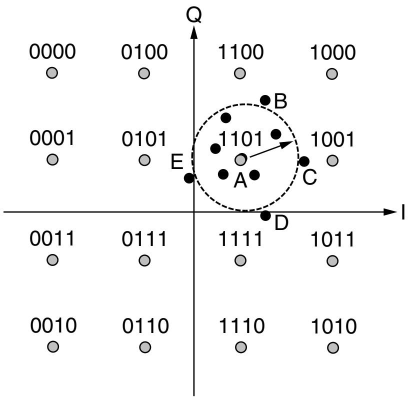
Figure 2-23. (a) QPSK. (b) QAM-16. (c) QAM-64.

4 bits per symbol. It is called **QAM-16**, where QAM stands for **Quadrature Amplitude Modulation**. Figure 2-23(c) is a still denser modulation scheme with 64 different combinations, so 6 bits can be transmitted per symbol. It is called **QAM-64**. Even higher-order QAMs are used too. As you might suspect from these constellations, it is easier to build electronics to produce symbols as a combination of values on each axis than as a combination of amplitude and phase values. That is why the patterns look like squares rather than concentric circles.

The constellations we have seen so far do not show how bits are assigned to symbols. When making the assignment, an important consideration is that a small burst of noise at the receiver not lead to many bit errors. This might happen if we assigned consecutive bit values to adjacent symbols. With QAM-16, for example, if one symbol stood for 0111 and the neighboring symbol stood for 1000, if the receiver mistakenly picks the adjacent symbol it will cause all of the bits to be wrong. A better solution is to map bits to symbols so that adjacent symbols differ in only 1 bit position. This mapping is called a **Gray code**. Fig. 2-24 shows a QAM-16 constellation that has been Gray coded. Now if the receiver decodes the symbol in error, it will make only a single bit error in the expected case that the decoded symbol is close to the transmitted symbol.

2.5.3 Frequency Division Multiplexing

The modulation schemes we have seen let us send one signal to convey bits along a wired or wireless link. However, economies of scale play an important role in how we use networks. It costs essentially the same amount of money to install and maintain a high-bandwidth transmission line as a low-bandwidth line between two different offices (i.e., the costs come from having to dig the trench and not from what kind of cable or fiber goes into it). Consequently, multiplexing schemes have been developed to share lines among many signals.



When 1101 is sent:

Point	Decodes as	Bit errors
A	1101	0
B	1100	1
C	1001	1
D	1111	1
E	0101	1

Figure 2-24. Gray-coded QAM-16.

FDM (**Frequency Division Multiplexing**) takes advantage of passband transmission to share a channel. It divides the spectrum into frequency bands, with each user having exclusive possession of some band in which to send their signal. AM radio broadcasting illustrates FDM. The allocated spectrum is about 1 MHz, roughly 500 to 1500 kHz. Different frequencies are allocated to different logical channels (stations), each operating in a portion of the spectrum, with the interchannel separation great enough to prevent interference.

For a more detailed example, in Fig. 2-25 we show three voice-grade telephone channels multiplexed using FDM. Filters limit the usable bandwidth to about 3100 Hz per voice-grade channel. When many channels are multiplexed together, 4000 Hz is allocated per channel. The excess is called a **guard band**. It keeps the channels well separated. First the voice channels are raised in frequency, each by a different amount. Then they can be combined because no two channels now occupy the same portion of the spectrum. Notice that even though there are gaps between the channels thanks to the guard bands, there is some overlap between adjacent channels. The overlap is there because real filters do not have ideal sharp edges. This means that a strong spike at the edge of one channel will be felt in the adjacent one as nonthermal noise.

This scheme has been used to multiplex calls in the telephone system for many years, but multiplexing in time is now preferred instead. However, FDM continues to be used in telephone networks, as well as cellular, terrestrial wireless, and satellite networks at a higher level of granularity.

When sending digital data, it is possible to divide the spectrum efficiently without using guard bands. In **OFDM** (**Orthogonal Frequency Division Multiplexing**), the channel bandwidth is divided into many subcarriers that independently send data (e.g., with QAM). The subcarriers are packed tightly together in the frequency domain. Thus, signals from each subcarrier extend into adjacent ones. However, as seen in Fig. 2-26, the frequency response of each subcarrier is

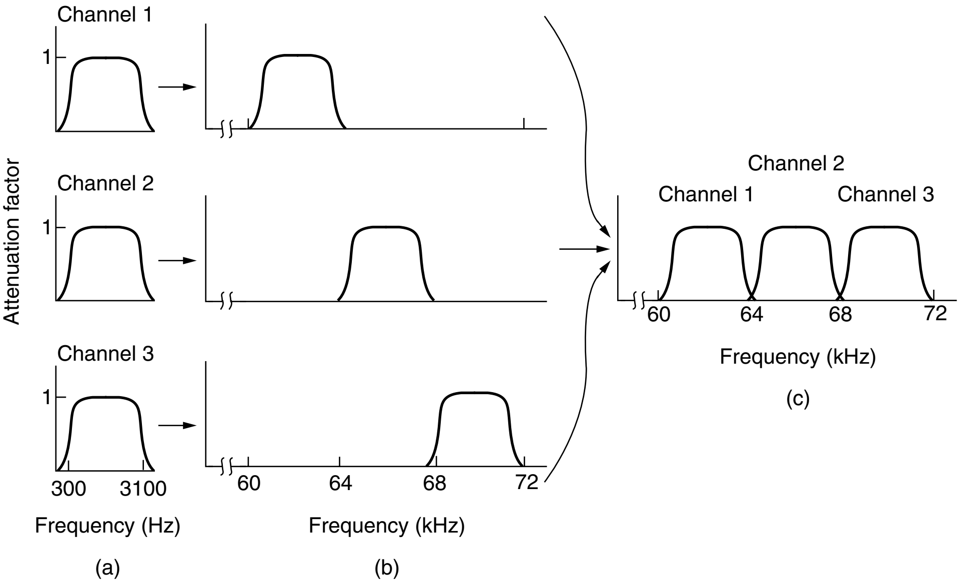


Figure 2-25. Frequency division multiplexing. (a) The original bandwidths. (b) The bandwidths raised in frequency. (c) The multiplexed channel.

designed so that it is zero at the center of the adjacent subcarriers. The subcarriers can therefore be sampled at their center frequencies without interference from their neighbors. To make this work, a guard time is needed to repeat a portion of the symbol signals in time so that they have the desired frequency response. However, this overhead is much less than is needed for many guard bands.

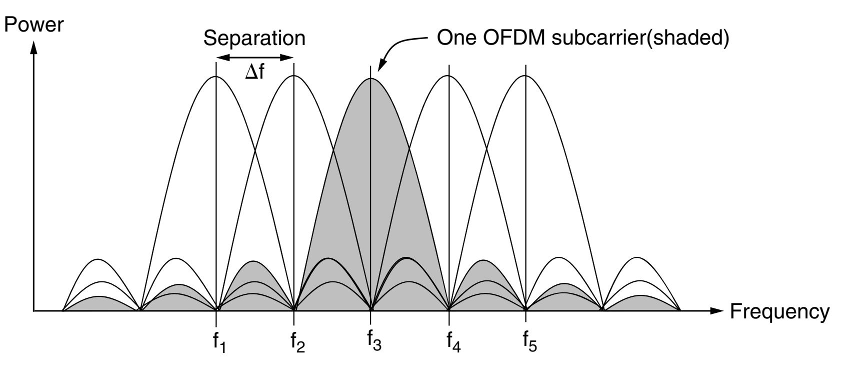


Figure 2-26. Orthogonal frequency division multiplexing (OFDM).

The idea of OFDM has been around for a long time, but it is only in the last decade that it has been widely adopted, following the realization that it is possible to implement OFDM efficiently in terms of a Fourier transform of digital data over all subcarriers (instead of separately modulating each subcarrier). OFDM is used in 802.11, cable networks and power line networking, and is planned for fourth-generation cellular systems. Usually, one high-rate stream of digital information is split into many low-rate streams that are transmitted on the subcarriers in parallel. This division is valuable because degradations of the channel are easier to cope with at the subcarrier level; some subcarriers may be very degraded and excluded in favor of subcarriers that are received well.

2.5.4 Time Division Multiplexing

An alternative to FDM is **TDM** (**Time Division Multiplexing**). Here, the users take turns (in a round-robin fashion), each one periodically getting the entire bandwidth for a little burst of time. An example of three streams being multiplexed with TDM is shown in Fig. 2-27. Bits from each input stream are taken in a fixed **time slot** and output to the aggregate stream. This stream runs at the sum rate of the individual streams. For this to work, the streams must be synchronized in time. Small intervals of **guard time** analogous to a frequency guard band may be added to accommodate small timing variations.

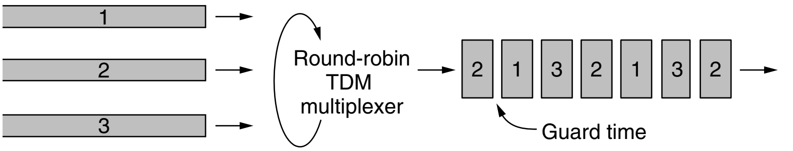


Figure 2-27. Time Division Multiplexing (TDM).

TDM is used widely as part of the telephone and cellular networks. To avoid one point of confusion, let us be clear that it is quite different from the alternative **STDM** (**Statistical Time Division Multiplexing**). The prefix ''statistical'' is added to indicate that the individual streams contribute to the multiplexed stream *not* on a fixed schedule, but according to the statistics of their demand. STDM is packet switching by another name.

2.5.5 Code Division Multiplexing

There is a third kind of multiplexing that works in a completely different way than FDM and TDM. **CDM** (**Code Division Multiplexing**) is a form of **spread spectrum** communication in which a narrowband signal is spread out over a wider frequency band. This can make it more tolerant of interference, as well as allowing multiple signals from different users to share the same frequency band. Because code division multiplexing is mostly used for the latter purpose it is commonly called **CDMA** (**Code Division Multiple Access**).

CDMA allows each station to transmit over the entire frequency spectrum all the time. Multiple simultaneous transmissions are separated using coding theory. Before getting into the algorithm, let us consider an analogy: an airport lounge with many pairs of people conversing. TDM is comparable to pairs of people in the room taking turns speaking. FDM is comparable to the pairs of people speaking at different pitches, some high-pitched and some low-pitched such that each pair can hold its own conversation at the same time as but independently of the others. CDMA is comparable to each pair of people talking at once, but in a different language. The French-speaking couple just hones in on the French, rejecting everything that is not French as noise. Thus, the key to CDMA is to be able to extract the desired signal while rejecting everything else as random noise. A somewhat simplified description of CDMA follows.

In CDMA, each bit time is subdivided into m short intervals called **chips**. Typically, there are 64 or 128 chips per bit, but in the example given here we will use 8 chips/bit for simplicity. Each station is assigned a unique m-bit code called a **chip sequence**. For pedagogical purposes, it is convenient to use a bipolar notation to write these codes as sequences of −1 and +1. We will show chip sequences in parentheses.

To transmit a 1 bit, a station sends its chip sequence. To transmit a 0 bit, it sends the negation of its chip sequence. No other patterns are permitted. Thus, for $m = 8$, if station A is assigned the chip sequence (−1 −1 −1 +1 +1 −1 +1 +1), it can send a 1 bit by transmiting the chip sequence and a 0 by transmitting (+1 +1 +1 −1 −1 +1 −1 −1). It is really signals with these voltage levels that are sent, but it is sufficient for us to think in terms of the sequences.

Increasing the amount of information to be sent from b bits/sec to mb chips/sec for each station means that the bandwidth needed for CDMA is greater by a factor of m than the bandwidth needed for a station not using CDMA (assuming no changes in the modulation or encoding techniques). If we have a 1-MHz band available for 100 stations, with FDM each one would have 10 kHz and could send at 10 kbps (assuming 1 bit per Hz). With CDMA, each station uses the full 1 MHz, so the chip rate is 100 chips per bit to spread the station's bit rate of 10 kbps across the channel.

In Fig. 2-28(a) and (b) we show the chip sequences assigned to four example stations and the signals that they represent. Each station has its own unique chip sequence. Let us use the symbol $\mathbf{S}$ to indicate the m-chip vector for station S, and $\overline{\mathbf{S}}$ for its negation. All chip sequences are pairwise **orthogonal**, by which we mean that the normalized inner product of any two distinct chip sequences, $\mathbf{S}$ and $\mathbf{T}$ (written as $\mathbf{S} \bullet \mathbf{T}$), is 0. It is known how to generate such orthogonal chip sequences using a method known as **Walsh codes**. In mathematical terms, orthogonality of the chip sequences can be expressed as follows:

$$\mathbf{S} \bullet \mathbf{T} \equiv \frac{1}{m} \sum_{i=1}^{m} S_i T_i = 0 \tag{2-5}$$

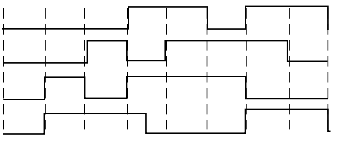
In plain English, as many pairs are the same as are different. This orthogonality property will prove crucial later. Note that if $\mathbf{S} \bullet \mathbf{T} = 0$, then $\mathbf{S} \bullet \overline{\mathbf{T}}$ is also 0. The normalized inner product of any chip sequence with itself is 1:

$$\mathbf{S} \bullet \mathbf{S} = \frac{1}{m} \sum_{i=1}^{m} S_i S_i = \frac{1}{m} \sum_{i=1}^{m} S_i^2 = \frac{1}{m} \sum_{i=1}^{m} (\pm 1)^2 = 1$$

This follows because each of the m terms in the inner product is 1, so the sum is m. Also note that $\mathbf{S} \bullet \overline{\mathbf{S}} = -1$.

A = (−1 −1 −1 +1 +1 −1 +1 +1)
B = (−1 −1 +1 −1 +1 +1 +1 −1)
C = (−1 +1 −1 +1 +1 +1 −1 −1)
D = (−1 +1 −1 −1 −1 −1 +1 −1)

(a)



(b)

$S_1 = C$ = (−1 +1 −1 +1 +1 +1 −1 −1)
$S_2 = B+C$ = (−2 0 0 0 +2 +2 0 −2)
$S_3 = A+\bar{B}$ = (0 0 −2 +2 0 −2 0 +2)
$S_4 = A+\bar{B}+C$ = (−1 +1 −3 +3 +1 −1 −1 +1)
$S_5 = A+B+C+D$ = (−4 0 −2 0 +2 0 +2 −2)
$S_6 = A+B+\bar{C}+D$ = (−2 −2 0 −2 0 −2 +4 0)

(c)

$S_1 \bullet C$ = [1+1−1+1+1+1−1−1]/8 = 1
$S_2 \bullet C$ = [2+0+0+0+2+2+0+2]/8 = 1
$S_3 \bullet C$ = [0+0+2+2+0−2+0−2]/8 = 0
$S_4 \bullet C$ = [1+1+3+3+1−1+1−1]/8 = 1
$S_5 \bullet C$ = [4+0+2+0+2+0−2+2]/8 = 1
$S_6 \bullet C$ = [2−2+0−2+0−2−4+0]/8 = −1

(d)

Figure 2-28. (a) Chip sequences for four stations. (b) Signals the sequences represent (c) Six examples of transmissions. (d) Recovery of station C's signal.

During each bit time, a station can transmit a 1 (by sending its chip sequence), it can transmit a 0 (by sending the negative of its chip sequence), or it can be silent and transmit nothing. We assume for now that all stations are synchronized in time, so all chip sequences begin at the same instant. When two or more stations transmit simultaneously, their bipolar sequences add linearly. For example, if in one chip period three stations output +1 and one station outputs −1, +2 will be received. One can think of this as signals that add as voltages superimposed on the channel: three stations output +1 V and one station outputs −1 V, so that 2 V is received. For instance, in Fig. 2-28(c) we see six examples of one or more stations transmitting 1 bit at the same time. In the first example, C transmits a 1 bit, so we just get C's chip sequence. In the second example, both B and C transmit 1 bits, so we get the sum of their bipolar chip sequences, namely:

$$(-1\ -1\ +1\ -1\ +1\ +1\ +1\ -1) + (-1\ +1\ -1\ +1\ +1\ +1\ -1\ -1) = (-2\ \ 0\ \ 0\ \ 0\ +2\ +2\ \ 0\ -2)$$

To recover the bit stream of an individual station, the receiver must know that station's chip sequence in advance. It does the recovery by computing the normalized inner product of the received chip sequence and the chip sequence of the station whose bit stream it is trying to recover. If the received chip sequence is $\mathbf{S}$ and the receiver is trying to listen to a station whose chip sequence is $\mathbf{C}$, it just computes the normalized inner product, $\mathbf{S} \bullet \mathbf{C}$.

To see why this works, just imagine that two stations, A and C, both transmit a 1 bit at the same time that B transmits a 0 bit, as is the case in the third example. The receiver sees the sum, $\mathbf{S} = \mathbf{A} + \overline{\mathbf{B}} + \mathbf{C}$, and computes

$$\mathbf{S} \bullet \mathbf{C} = (\mathbf{A} + \overline{\mathbf{B}} + \mathbf{C}) \bullet \mathbf{C} = \mathbf{A} \bullet \mathbf{C} + \overline{\mathbf{B}} \bullet \mathbf{C} + \mathbf{C} \bullet \mathbf{C} = 0 + 0 + 1 = 1$$

The first two terms vanish because all pairs of chip sequences have been carefully chosen to be orthogonal, as shown in Eq. (2-5). Now it should be clear why this property must be imposed on the chip sequences.

To make the decoding process more concrete, we show six examples in Fig. 2-28(d). Suppose that the receiver is interested in extracting the bit sent by station C from each of the six signals S_1 through S_6. It calculates the bit by summing the pairwise products of the received **S** and the **C** vector of Fig. 2-28(a) and then taking 1/8 of the result (since $m = 8$ here). The examples include cases where C is silent, sends a 1 bit, and sends a 0 bit, individually and in combination with other transmissions. As shown, the correct bit is decoded each time. It is just like speaking French.

In principle, given enough computing capacity, the receiver can listen to all the senders at once by running the decoding algorithm for each of them in parallel. In real life, suffice it to say that this is easier said than done, and it is useful to know which senders might be transmitting.

In the ideal, noiseless CDMA system we have studied here, the number of stations that send concurrently can be made arbitrarily large by using longer chip sequences. For 2^n stations, Walsh codes can provide 2^n orthogonal chip sequences of length 2^n. However, one significant limitation is that we have assumed that all the chips are synchronized in time at the receiver. This synchronization is not even approximately true in some applications, such as cellular networks (in which CDMA has been widely deployed starting in the 1990s). It leads to different designs. We will return to this topic later in the chapter and describe how asynchronous CDMA differs from synchronous CDMA.

As well as cellular networks, CDMA is used by satellites and cable networks. We have glossed over many complicating factors in this brief introduction. Engineers who want to gain a deep understanding of CDMA should read Viterbi (1995) and Lee and Miller (1998). These references require quite a bit of background in communication engineering, however.

2.6 THE PUBLIC SWITCHED TELEPHONE NETWORK

When two computers owned by the same company or organization and located close to each other need to communicate, it is often easiest just to run a cable between them. LANs work this way. However, when the distances are large or there are many computers or the cables have to pass through a public road or other public right of way, the costs of running private cables are usually prohibitive.

Furthermore, in just about every country in the world, stringing private transmission lines across (or underneath) public property is also illegal. Consequently, the network designers must rely on the existing telecommunication facilities.

These facilities, especially the **PSTN** (**Public Switched Telephone Network**), were usually designed many years ago, with a completely different goal in mind: transmitting the human voice in a more-or-less recognizable form. Their suitability for use in computer-computer communication is often marginal at best. To see the size of the problem, consider that a cheap commodity cable running between two computers can transfer data at 1 Gbps or more. In contrast, typical ADSL, the blazingly fast alternative to a telephone modem, runs at around 1 Mbps. The difference between the two is the difference between cruising in an airplane and taking a leisurely stroll.

Nonetheless, the telephone system is tightly intertwined with (wide area) computer networks, so it is worth devoting some time to study it in detail. The limiting factor for networking purposes turns out to be the ''last mile'' over which customers connect, not the trunks and switches inside the telephone network. This situation is changing with the gradual rollout of fiber and digital technology at the edge of the network, but it will take time and money. During the long wait, computer systems designers used to working with systems that give at least three orders of magnitude better performance have devoted much time and effort to figure out how to use the telephone network efficiently.

In the following sections we will describe the telephone system and show how it works. For additional information about the innards of the telephone system see Bellamy (2000).

2.6.1 Structure of the Telephone System

Soon after Alexander Graham Bell patented the telephone in 1876 (just a few hours ahead of his rival, Elisha Gray), there was an enormous demand for his new invention. The initial market was for the sale of telephones, which came in pairs. It was up to the customer to string a single wire between them. If a telephone owner wanted to talk to *n* other telephone owners, separate wires had to be strung to all *n* houses. Within a year, the cities were covered with wires passing over houses and trees in a wild jumble. It became immediately obvious that the model of connecting every telephone to every other telephone, as shown in Fig. 2-29(a), was not going to work.

To his credit, Bell saw this problem early on and formed the Bell Telephone Company, which opened its first switching office (in New Haven, Connecticut) in 1878. The company ran a wire to each customer's house or office. To make a call, the customer would crank the phone to make a ringing sound in the telephone company office to attract the attention of an operator, who would then manually connect the caller to the callee by using a short jumper cable to connect the caller to the callee. The model of a single switching office is illustrated in Fig. 2-29(b).

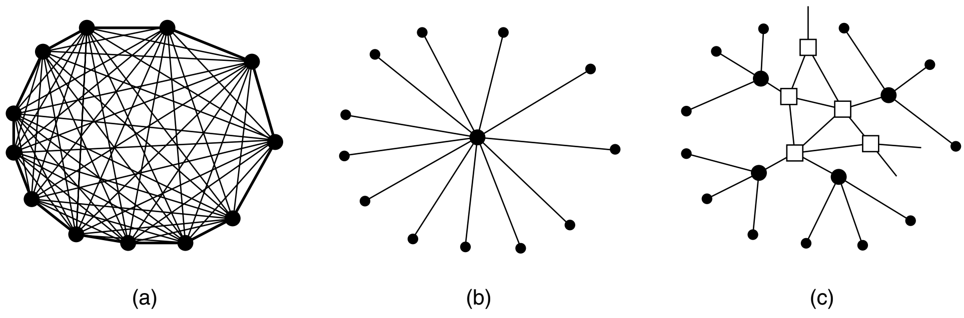


Figure 2-29. (a) Fully interconnected network. (b) Centralized switch. (c) Two-level hierarchy.

Pretty soon, Bell System switching offices were springing up everywhere and people wanted to make long-distance calls between cities, so the Bell System began to connect the switching offices. The original problem soon returned: to connect every switching office to every other switching office by means of a wire between them quickly became unmanageable, so second-level switching offices were invented. After a while, multiple second-level offices were needed, as illustrated in Fig. 2-29(c). Eventually, the hierarchy grew to five levels.

By 1890, the three major parts of the telephone system were in place: the switching offices, the wires between the customers and the switching offices (by now balanced, insulated, twisted pairs instead of open wires with an earth return), and the long-distance connections between the switching offices. For a short technical history of the telephone system, see Hawley (1991).

While there have been improvements in all three areas since then, the basic Bell System model has remained essentially intact for over 100 years. The following description is highly simplified but gives the essential flavor nevertheless. Each telephone has two copper wires coming out of it that go directly to the telephone company's nearest **end office** (also called a **local central office**). The distance is typically 1 to 10 km, being shorter in cities than in rural areas. In the United States alone there are about 22,000 end offices. The two-wire connections between each subscriber's telephone and the end office are known in the trade as the **local loop**. If the world's local loops were stretched out end to end, they would extend to the moon and back 1000 times.

At one time, 80% of AT&T's capital value was the copper in the local loops. AT&T was then, in effect, the world's largest copper mine. Fortunately, this fact was not well known in the investment community. Had it been known, some corporate raider might have bought AT&T, ended all telephone service in the United States, ripped out all the wire, and sold it to a copper refiner for a quick payback.

If a subscriber attached to a given end office calls another subscriber attached to the same end office, the switching mechanism within the office sets up a direct electrical connection between the two local loops. This connection remains intact for the duration of the call.

If the called telephone is attached to another end office, a different procedure has to be used. Each end office has a number of outgoing lines to one or more nearby switching centers, called **toll offices** (or, if they are within the same local area, **tandem offices**). These lines are called **toll connecting trunks**. The number of different kinds of switching centers and their topology varies from country to country depending on the country's telephone density.

If both the caller's and callee's end offices happen to have a toll connecting trunk to the same toll office (a likely occurrence if they are relatively close by), the connection may be established within the toll office. A telephone network consisting only of telephones (the small dots), end offices (the large dots), and toll offices (the squares) is shown in Fig. 2-29(c).

If the caller and callee do not have a toll office in common, a path will have to be established between two toll offices. The toll offices communicate with each other via high-bandwidth **intertoll trunks** (also called **interoffice trunks**). Prior to the 1984 breakup of AT&T, the U.S. telephone system used hierarchical routing to find a path, going to higher levels of the hierarchy until there was a switching office in common. This was then replaced with more flexible, nonhierarchical routing. Figure 2-30 shows how a long-distance connection might be routed.

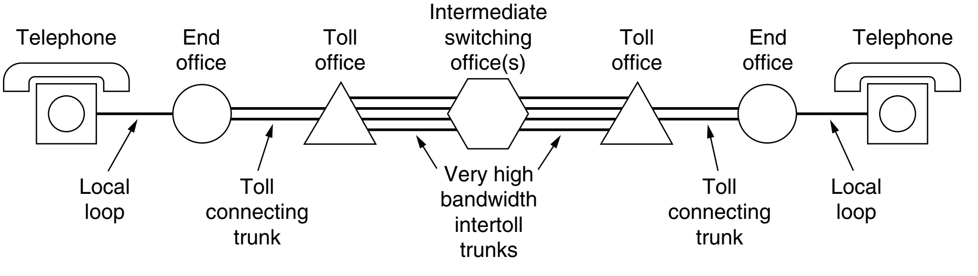


Figure 2-30. A typical circuit route for a long-distance call.

A variety of transmission media are used for telecommunication. Unlike modern office buildings, where the wiring is commonly Category 5, local loops to homes mostly consist of Category 3 twisted pairs, with fiber just starting to appear. Between switching offices, coaxial cables, microwaves, and especially fiber optics are widely used.

In the past, transmission throughout the telephone system was analog, with the actual voice signal being transmitted as an electrical voltage from source to destination. With the advent of fiber optics, digital electronics, and computers, all the trunks and switches are now digital, leaving the local loop as the last piece of analog technology in the system. Digital transmission is preferred because it is not necessary to accurately reproduce an analog waveform after it has passed through many amplifiers on a long call. Being able to correctly distinguish a 0 from a 1 is enough. This property makes digital transmission more reliable than analog. It is also cheaper and easier to maintain.

In summary, the telephone system consists of three major components:

1. Local loops (analog twisted pairs going to houses and businesses).
2. Trunks (digital fiber optic links connecting the switching offices).
3. Switching offices (where calls are moved from one trunk to another).

After a short digression on the politics of telephones, we will come back to each of these three components in some detail. The local loops provide everyone access to the whole system, so they are critical. Unfortunately, they are also the weakest link in the system. For the long-haul trunks, the main issue is how to collect multiple calls together and send them out over the same fiber. This calls for multiplexing, and we apply FDM and TDM to do it. Finally, there are two fundamentally different ways of doing switching; we will look at both.

2.6.2 The Politics of Telephones

For decades prior to 1984, the Bell System provided both local and long-distance service throughout most of the United States. In the 1970s, the U.S. Federal Government came to believe that this was an illegal monopoly and sued to break it up. The government won, and on January 1, 1984, AT&T was broken up into AT&T Long Lines, 23 **BOCs** (**Bell Operating Companies**), and a few other pieces. The 23 BOCs were grouped into seven regional BOCs (RBOCs) to make them economically viable. The entire nature of telecommunication in the United States was changed overnight by court order (*not* by an act of Congress).

The exact specifications of the divestiture were described in the so-called **MFJ** (**Modified Final Judgment**), an oxymoron if ever there was one—if the judgment could be modified, it clearly was not final. This event led to increased competition, better service, and lower long-distance rates for consumers and businesses. However, prices for local service rose as the cross subsidies from long-distance calling were eliminated and local service had to become self supporting. Many other countries have now introduced competition along similar lines.

Of direct relevance to our studies is that the new competitive framework caused a key technical feature to be added to the architecture of the telephone network. To make it clear who could do what, the United States was divided up into 164 **LATAs** (**Local Access and Transport Areas**). Very roughly, a LATA is about as big as the area covered by one area code. Within each LATA, there was one **LEC** (**Local Exchange Carrier**) with a monopoly on traditional telephone service within its area. The most important LECs were the BOCs, although some LATAs contained one or more of the 1500 independent telephone companies operating as LECs.

The new feature was that all inter-LATA traffic was handled by a different kind of company, an **IXC** (**IntereXchange Carrier**). Originally, AT&T Long Lines was the only serious IXC, but now there are well-established competitors such as Verizon and Sprint in the IXC business. One of the concerns at the breakup was to ensure that all the IXCs would be treated equally in terms of line quality, tariffs, and the number of digits their customers would have to dial to use them. The way this is handled is illustrated in Fig. 2-31. Here we see three example LATAs, each with several end offices. LATAs 2 and 3 also have a small hierarchy with tandem offices (intra-LATA toll offices).

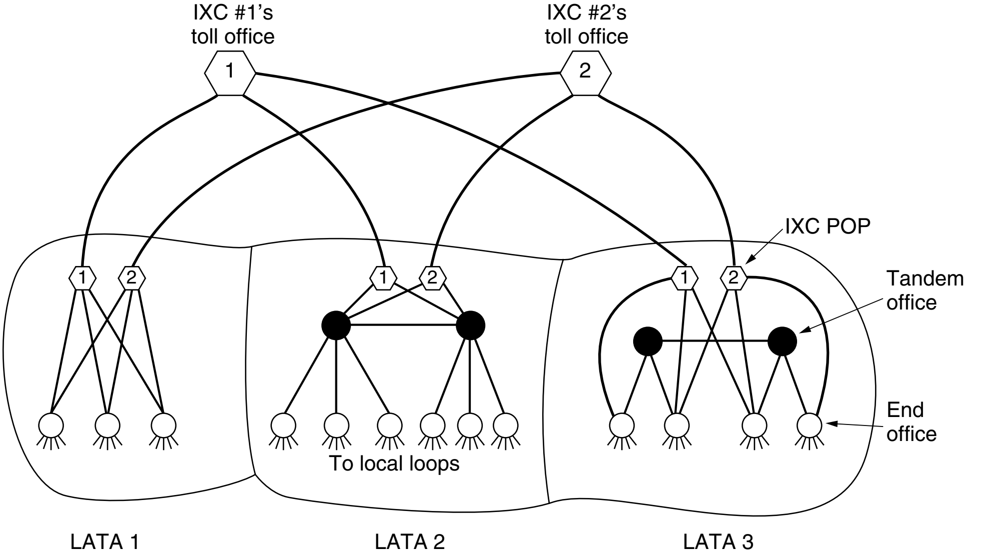


Figure 2-31. The relationship of LATAs, LECs, and IXCs. All the circles are LEC switching offices. Each hexagon belongs to the IXC whose number is in it.

Any IXC that wishes to handle calls originating in a LATA can build a switching office called a **POP** (**Point of Presence**) there. The LEC is required to connect each IXC to every end office, either directly, as in LATAs 1 and 3, or indirectly, as in LATA 2. Furthermore, the terms of the connection, both technical and financial, must be identical for all IXCs. This requirement enables, a subscriber in, say, LATA 1, to choose which IXC to use for calling subscribers in LATA 3.

As part of the MFJ, the IXCs were forbidden to offer local telephone service and the LECs were forbidden to offer inter-LATA telephone service, although both were free to enter any other business, such as operating fried chicken restaurants. In 1984, that was a fairly unambiguous statement. Unfortunately, technology has a funny way of making the law obsolete. Neither cable television nor mobile phones were covered by the agreement. As cable television went from one way to two way and mobile phones exploded in popularity, both LECs and IXCs began buying up or merging with cable and mobile operators.

By 1995, Congress saw that trying to maintain a distinction between the various kinds of companies was no longer tenable and drafted a bill to preserve accessibility for competition but allow cable TV companies, local telephone companies, long-distance carriers, and mobile operators to enter one another's businesses. The idea was that any company could then offer its customers a single integrated package containing cable TV, telephone, and information services and that different companies would compete on service and price. The bill was enacted into law in February 1996 as a major overhaul of telecommunications regulation. As a result, some BOCs became IXCs and some other companies, such as cable television operators, began offering local telephone service in competition with the LECs.

One interesting property of the 1996 law is the requirement that LECs implement **local number portability**. This means that a customer can change local telephone companies without having to get a new telephone number. Portability for mobile phone numbers (and between fixed and mobile lines) followed suit in 2003. These provisions removed a huge hurdle for many people, making them much more inclined to switch LECs. As a result, the U.S. telecommunications landscape became much more competitive, and other countries have followed suit. Often other countries wait to see how this kind of experiment works out in the U.S. If it works well, they do the same thing; if it works badly, they try something else.

2.6.3 The Local Loop: Modems, ADSL, and Fiber

It is now time to start our detailed study of how the telephone system works. Let us begin with the part that most people are familiar with: the two-wire local loop coming from a telephone company end office into houses. The local loop is also frequently referred to as the ''last mile,'' although the length can be up to several miles. It has carried analog information for over 100 years and is likely to continue doing so for some years to come, due to the high cost of converting to digital.

Much effort has been devoted to squeezing data networking out of the copper local loops that are already deployed. Telephone modems send digital data between computers over the narrow channel the telephone network provides for a voice call. They were once widely used, but have been largely displaced by broadband technologies such as ADSL that. reuse the local loop to send digital data from a customer to the end office, where they are siphoned off to the Internet.

Both modems and ADSL must deal with the limitations of old local loops: relatively narrow bandwidth, attenuation and distortion of signals, and susceptibility to electrical noise such as crosstalk.

In some places, the local loop has been modernized by installing optical fiber to (or very close to) the home. Fiber is the way of the future. These installations support computer networks from the ground up, with the local loop having ample bandwidth for data services. The limiting factor is what people will pay, not the physics of the local loop.

In this section we will study the local loop, both old and new. We will cover telephone modems, ADSL, and fiber to the home.

Telephone Modems

To send bits over the local loop, or any other physical channel for that matter, they must be converted to analog signals that can be transmitted over the channel. This conversion is accomplished using the methods for digital modulation that we studied in the previous section. At the other end of the channel, the analog signal is converted back to bits.

A device that converts between a stream of digital bits and an analog signal that represents the bits is called a **modem**, which is short for "*mo*dulator *dem*odulator." Modems come in many varieties: telephone modems, DSL modems, cable modems, wireless modems, etc. The modem may be built into the computer (which is now common for telephone modems) or be a separate box (which is common for DSL and cable modems). Logically, the modem is inserted between the (digital) computer and the (analog) telephone system, as seen in Fig. 2-32.

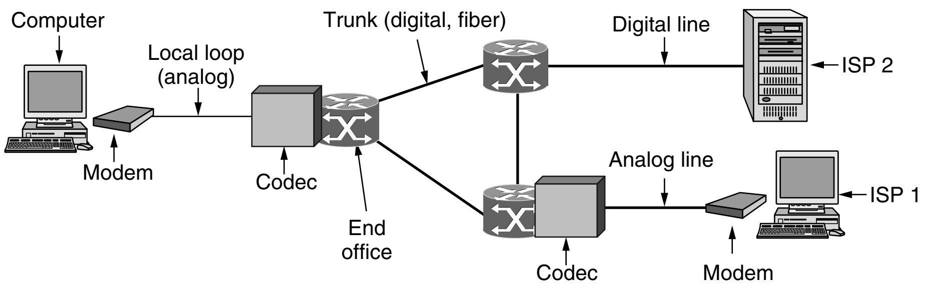


Figure 2-32. The use of both analog and digital transmission for a computer-to-computer call. Conversion is done by the modems and codecs.

Telephone modems are used to send bits between two computers over a voice-grade telephone line, in place of the conversation that usually fills the line. The main difficulty in doing so is that a voice-grade telephone line is limited to 3100 Hz, about what is sufficient to carry a conversation. This bandwidth is more than four orders of magnitude less than the bandwidth that is used for Ethernet or

802.11 (WiFi). Unsurprisingly, the data rates of telephone modems are also four orders of magnitude less than that of Ethernet and 802.11.

Let us run the numbers to see why this is the case. The Nyquist theorem tells us that even with a perfect 3000-Hz line (which a telephone line is decidedly not), there is no point in sending symbols at a rate faster than 6000 baud. In practice, most modems send at a rate of 2400 symbols/sec, or 2400 baud, and focus on getting multiple bits per symbol while allowing traffic in both directions at the same time (by using different frequencies for different directions).

The humble 2400-bps modem uses 0 volts for a logical 0 and 1 volt for a logical 1, with 1 bit per symbol. One step up, it can use four different symbols, as in the four phases of QPSK, so with 2 bits/symbol it can get a data rate of 4800 bps.

A long progression of higher rates has been achieved as technology has improved. Higher rates require a larger set of symbols or **constellation**. With many symbols, even a small amount of noise in the detected amplitude or phase can result in an error. To reduce the chance of errors, standards for the higher-speed modems use some of the symbols for error correction. The schemes are known as **TCM** (**Trellis Coded Modulation**) (Ungerboeck, 1987).

The **V.32** modem standard uses 32 constellation points to transmit 4 data bits and 1 check bit per symbol at 2400 baud to achieve 9600 bps with error correction. The next step above 9600 bps is 14,400 bps. It is called **V.32 bis** and transmits 6 data bits and 1 check bit per symbol at 2400 baud. Then comes **V.34**, which achieves 28,800 bps by transmitting 12 data bits/symbol at 2400 baud. The constellation now has thousands of points. The final modem in this series is **V.34 bis** which uses 14 data bits/symbol at 2400 baud to achieve 33,600 bps.

Why stop here? The reason that standard modems stop at 33,600 is that the Shannon limit for the telephone system is about 35 kbps based on the average length of local loops and the quality of these lines. Going faster than this would violate the laws of physics (department of thermodynamics).

However, there is one way we can change the situation. At the telephone company end office, the data are converted to digital form for transmission within the telephone network (the core of the telephone network converted from analog to digital long ago). The 35-kbps limit is for the situation in which there are two local loops, one at each end. Each of these adds noise to the signal. If we could get rid of one of these local loops, we would increase the SNR and the maximum rate would be doubled.

This approach is how 56-kbps modems are made to work. One end, typically an ISP, gets a high-quality digital feed from the nearest end office. Thus, when one end of the connection is a high-quality signal, as it is with most ISPs now, the maximum data rate can be as high as 70 kbps. Between two home users with modems and analog lines, the maximum is still 33.6 kbps.

The reason that 56-kbps modems (rather than 70-kbps modems) are in use has to do with the Nyquist theorem. A telephone channel is carried inside the telephone system as digital samples. Each telephone channel is 4000 Hz wide when the guard bands are included. The number of samples per second needed to reconstruct it is thus 8000. The number of bits per sample in the U.S. is 8, one of which may be used for control purposes, allowing 56,000 bits/sec of user data. In Europe, all 8 bits are available to users, so 64,000-bit/sec modems could have been used, but to get international agreement on a standard, 56,000 was chosen.

The end result is the **V.90** and **V.92** modem standards. They provide for a 56-kbps downstream channel (ISP to user) and a 33.6-kbps and 48-kbps upstream channel (user to ISP), respectively. The asymmetry is because there is usually more data transported from the ISP to the user than the other way. It also means that more of the limited bandwidth can be allocated to the downstream channel to increase the chances of it actually working at 56 kbps.

Digital Subscriber Lines

When the telephone industry finally got to 56 kbps, it patted itself on the back for a job well done. Meanwhile, the cable TV industry was offering speeds up to 10 Mbps on shared cables. As Internet access became an increasingly important part of their business, the telephone companies (LECs) began to realize they needed a more competitive product. Their answer was to offer new digital services over the local loop.

Initially, there were many overlapping high-speed offerings, all under the general name of **xDSL** (**Digital Subscriber Line**), for various *x*. Services with more bandwidth than standard telephone service are sometimes called **broadband**, although the term really is more of a marketing concept than a specific technical concept. Later, we will discuss what has become the most popular of these services, **ADSL** (**Asymmetric DSL**). We will also use the term DSL or xDSL as shorthand for all flavors.

The reason that modems are so slow is that telephones were invented for carrying the human voice and the entire system has been carefully optimized for this purpose. Data have always been stepchildren. At the point where each local loop terminates in the end office, the wire runs through a filter that attenuates all frequencies below 300 Hz and above 3400 Hz. The cutoff is not sharp—300 Hz and 3400 Hz are the 3-dB points—so the bandwidth is usually quoted as 4000 Hz even though the distance between the 3 dB points is 3100 Hz. Data on the wire are thus also restricted to this narrow band.

The trick that makes xDSL work is that when a customer subscribes to it, the incoming line is connected to a different kind of switch, one that does not have this filter, thus making the entire capacity of the local loop available. The limiting factor then becomes the physics of the local loop, which supports roughly 1 MHz, not the artificial 3100 Hz bandwidth created by the filter.

Unfortunately, the capacity of the local loop falls rather quickly with distance from the end office as the signal is increasingly degraded along the wire. It also depends on the thickness and general quality of the twisted pair. A plot of the potential bandwidth as a function of distance is given in Fig. 2-33. This figure assumes that all the other factors are optimal (new wires, modest bundles, etc.).

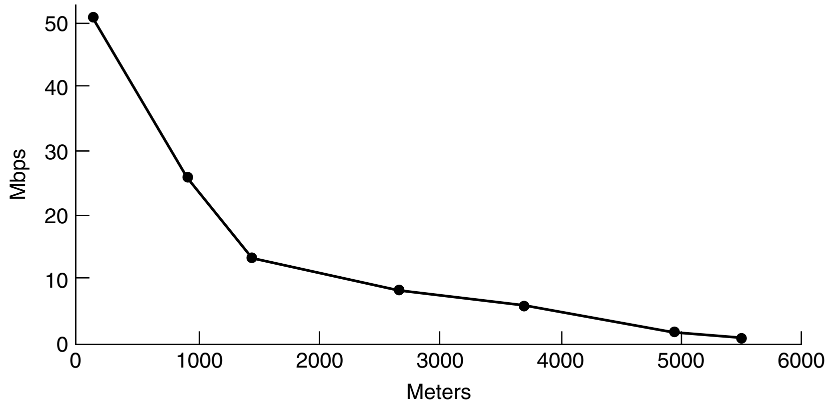


Figure 2-33. Bandwidth versus distance over Category 3 UTP for DSL.

The implication of this figure creates a problem for the telephone company. When it picks a speed to offer, it is simultaneously picking a radius from its end offices beyond which the service cannot be offered. This means that when distant customers try to sign up for the service, they may be told "Thanks a lot for your interest, but you live 100 meters too far from the nearest end office to get this service. Could you please move?" The lower the chosen speed is, the larger the radius and the more customers are covered. But the lower the speed, the less attractive the service is and the fewer the people who will be willing to pay for it. This is where business meets technology.

The xDSL services have all been designed with certain goals in mind. First, the services must work over the existing Category 3 twisted pair local loops. Second, they must not affect customers' existing telephones and fax machines. Third, they must be much faster than 56 kbps. Fourth, they should be always on, with just a monthly charge and no per-minute charge.

To meet the technical goals, the available 1.1 MHz spectrum on the local loop is divided into 256 independent channels of 4312.5 Hz each. This arrangement is shown in Fig. 2-34. The OFDM scheme, which we saw in the previous section, is used to send data over these channels, though it is often called **DMT** (**Discrete MultiTone**) in the context of ADSL. Channel 0 is used for **POTS** (**Plain Old Telephone Service**). Channels 1–5 are not used, to keep the voice and data signals from interfering with each other. Of the remaining 250 channels, one is used for upstream control and one is used for downstream control. The rest are available for user data.

In principle, each of the remaining channels can be used for a full-duplex data stream, but harmonics, crosstalk, and other effects keep practical systems well

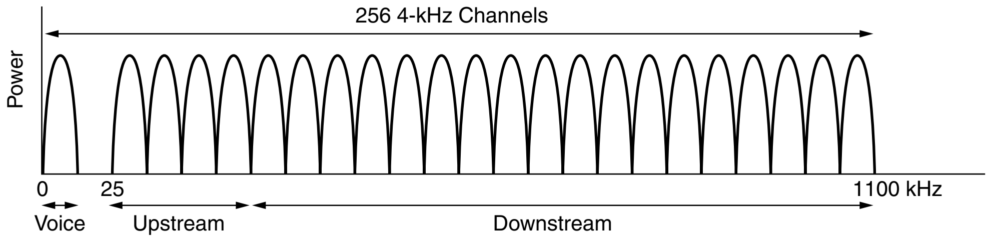


Figure 2-34. Operation of ADSL using discrete multitone modulation.

below the theoretical limit. It is up to the provider to determine how many channels are used for upstream and how many for downstream. A 50/50 mix of upstream and downstream is technically possible, but most providers allocate something like 80–90% of the bandwidth to the downstream channel since most users download more data than they upload. This choice gives rise to the "A" in ADSL. A common split is 32 channels for upstream and the rest downstream. It is also possible to have a few of the highest upstream channels be bidirectional for increased bandwidth, although making this optimization requires adding a special circuit to cancel echoes.

The international ADSL standard, known as **G.dmt**, was approved in 1999. It allows speeds of as much as 8 Mbps downstream and 1 Mbps upstream. It was superseded by a second generation in 2002, called ADSL2, with various improvements to allow speeds of as much as 12 Mbps downstream and 1 Mbps upstream. Now we have ADSL2+, which doubles the downstream speed to 24 Mbps by doubling the bandwidth to use 2.2 MHz over the twisted pair.

However, the numbers quoted here are best-case speeds for good lines close (within 1 to 2 km) to the exchange. Few lines support these rates, and few providers offer these speeds. Typically, providers offer something like 1 Mbps downstream and 256 kbps upstream (standard service), 4 Mbps downstream and 1 Mbps upstream (improved service), and 8 Mbps downstream and 2 Mbps upstream (premium service).

Within each channel, QAM modulation is used at a rate of roughly 4000 symbols/sec. The line quality in each channel is constantly monitored and the data rate is adjusted by using a larger or smaller constellation, like those in Fig. 2-23. Different channels may have different data rates, with up to 15 bits per symbol sent on a channel with a high SNR, and down to 2, 1, or no bits per symbol sent on a channel with a low SNR depending on the standard.

A typical ADSL arrangement is shown in Fig. 2-35. In this scheme, a telephone company technician must install a **NID** (**Network Interface Device**) on the customer's premises. This small plastic box marks the end of the telephone company's property and the start of the customer's property. Close to the NID (or sometimes combined with it) is a **splitter**, an analog filter that separates the

0–4000-Hz band used by POTS from the data. The POTS signal is routed to the existing telephone or fax machine. The data signal is routed to an ADSL modem, which uses digital signal processing to implement OFDM. Since most ADSL modems are external, the computer must be connected to them at high speed. Usually, this is done using Ethernet, a USB cable, or 802.11.

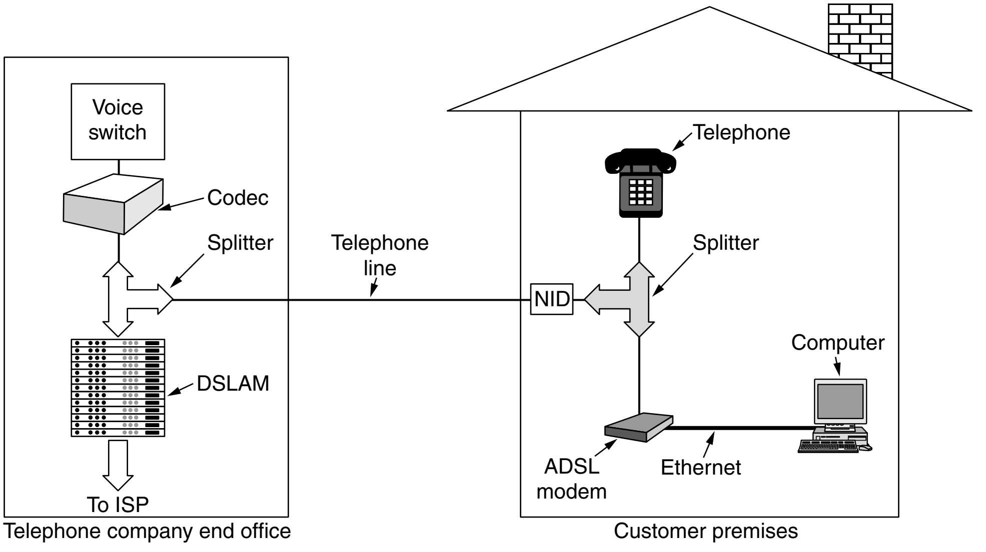


Figure 2-35. A typical ADSL equipment configuration.

At the other end of the wire, on the end office side, a corresponding splitter is installed. Here, the voice portion of the signal is filtered out and sent to the normal voice switch. The signal above 26 kHz is routed to a new kind of device called a **DSLAM** (**Digital Subscriber Line Access Multiplexer**), which contains the same kind of digital signal processor as the ADSL modem. Once the bits have been recovered from the signal, packets are formed and sent off to the ISP.

This complete separation between the voice system and ADSL makes it relatively easy for a telephone company to deploy ADSL. All that is needed is buying a DSLAM and splitter and attaching the ADSL subscribers to the splitter. Other high-bandwidth services (e.g., ISDN) require much greater changes to the existing switching equipment.

One disadvantage of the design of Fig. 2-35 is the need for a NID and splitter on the customer's premises. Installing these can only be done by a telephone company technician, necessitating an expensive "truck roll" (i.e., sending a technician to the customer's premises). Therefore, an alternative, splitterless design, informally called **G.lite**, has also been standardized. It is the same as Fig. 2-35 but without the customer's splitter. The existing telephone line is used as is. The only difference is that a microfilter has to be inserted into each telephone jack between the telephone or ADSL modem and the wire. The microfilter for the telephone is a low-pass filter eliminating frequencies above 3400 Hz; the microfilter for the ADSL modem is a high-pass filter eliminating frequencies below 26 kHz. However, this system is not as reliable as having a splitter, so G.lite can be used only up to 1.5 Mbps (versus 8 Mbps for ADSL with a splitter). For more information about ADSL, see Starr (2003).

Fiber To The Home

Deployed copper local loops limit the performance of ADSL and telephone modems. To let them provide faster and better network services, telephone companies are upgrading local loops at every opportunity by installing optical fiber all the way to houses and offices. The result is called **FttH** (**Fiber To The Home**). While FttH technology has been available for some time, deployments only began to take off in 2005 with growth in the demand for high-speed Internet from customers used to DSL and cable who wanted to download movies. Around 4% of U.S. houses are now connected to FttH with Internet access speeds of up to 100 Mbps.

Several variations of the form "FttX" (where *X* stands for the basement, curb, or neighborhood) exist. They are used to note that the fiber deployment may reach close to the house. In this case, copper (twisted pair or coaxial cable) provides fast enough speeds over the last short distance. The choice of how far to lay the fiber is an economic one, balancing cost with expected revenue. In any case, the point is that optical fiber has crossed the traditional barrier of the "last mile." We will focus on FttH in our discussion.

Like the copper wires before it, the fiber local loop is passive. This means no powered equipment is required to amplify or otherwise process signals. The fiber simply carries signals between the home and the end office. This in turn reduces cost and improves reliability.

Usually, the fibers from the houses are joined together so that only a single fiber reaches the end office per group of up to 100 houses. In the downstream direction, optical splitters divide the signal from the end office so that it reaches all the houses. Encryption is needed for security if only one house should be able to decode the signal. In the upstream direction, optical combiners merge the signals from the houses into a single signal that is received at the end office.

This architecture is called a **PON** (**Passive Optical Network**), and it is shown in Fig. 2-36. It is common to use one wavelength shared between all the houses for downstream transmission, and another wavelength for upstream transmission.

Even with the splitting, the tremendous bandwidth and low attenuation of fiber mean that PONs can provide high rates to users over distances of up to 20 km. The actual data rates and other details depend on the type of PON. Two kinds are common. **GPONs** (**Gigabit-capable PONs**) come from the world of telecommunications, so they are defined by an ITU standard. **EPONs** (**Ethernet PONs**)

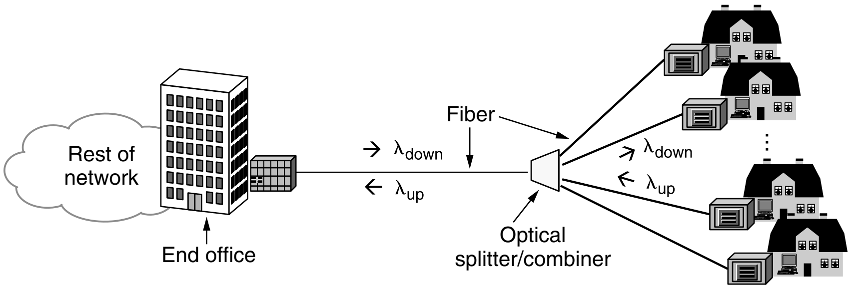


Figure 2-36. Passive optical network for Fiber To The Home.

are more in tune with the world of networking, so they are defined by an IEEE standard. Both run at around a gigabit and can carry traffic for different services, including Internet, video, and voice. For example, GPONs provide 2.4 Gbps downstream and 1.2 or 2.4 Gbps upstream.

Some protocol is needed to share the capacity of the single fiber at the end office between the different houses. The downstream direction is easy. The end office can send messages to each different house in whatever order it likes. In the upstream direction, however, messages from different houses cannot be sent at the same time, or different signals would collide. The houses also cannot hear each other's transmissions so they cannot listen before transmitting. The solution is that equipment at the houses requests and is granted time slots to use by equipment in the end office. For this to work, there is a ranging process to adjust the transmission times from the houses so that all the signals received at the end office are synchronized. The design is similar to cable modems, which we cover later in this chapter. For more information on the future of PONs, see Grobe and Elbers (2008).

2.6.4 Trunks and Multiplexing

Trunks in the telephone network are not only much faster than the local loops, they are different in two other respects. The core of the telephone network carries digital information, not analog information; that is, bits not voice. This necessitates a conversion at the end office to digital form for transmission over the long-haul trunks. The trunks carry thousands, even millions, of calls simultaneously. This sharing is important for achieving economies of scale, since it costs essentially the same amount of money to install and maintain a high-bandwidth trunk as a low-bandwidth trunk between two switching offices. It is accomplished with versions of TDM and FDM multiplexing.

Below we will briefly examine how voice signals are digitized so that they can be transported by the telephone network. After that, we will see how TDM is used to carry bits on trunks, including the TDM system used for fiber optics

(SONET). Then we will turn to FDM as it is applied to fiber optics, which is called wavelength division multiplexing.

Digitizing Voice Signals

Early in the development of the telephone network, the core handled voice calls as analog information. FDM techniques were used for many years to multiplex 4000-Hz voice channels (comprised of 3100 Hz plus guard bands) into larger and larger units. For example, 12 calls in the 60 kHz–to–108 kHz band is known as a **group** and five groups (a total of 60 calls) are known as a **supergroup**, and so on. These FDM methods are still used over some copper wires and microwave channels. However, FDM requires analog circuitry and is not amenable to being done by a computer. In contrast, TDM can be handled entirely by digital electronics, so it has become far more widespread in recent years. Since TDM can only be used for digital data and the local loops produce analog signals, a conversion is needed from analog to digital in the end office, where all the individual local loops come together to be combined onto outgoing trunks.

The analog signals are digitized in the end office by a device called a **codec** (short for "*co*der-*dec*oder"). The codec makes 8000 samples per second (125 μsec/sample) because the Nyquist theorem says that this is sufficient to capture all the information from the 4-kHz telephone channel bandwidth. At a lower sampling rate, information would be lost; at a higher one, no extra information would be gained. Each sample of the amplitude of the signal is quantized to an 8-bit number.

This technique is called **PCM** (**Pulse Code Modulation**). It forms the heart of the modern telephone system. As a consequence, virtually all time intervals within the telephone system are multiples of 125 μsec. The standard uncompressed data rate for a voice-grade telephone call is thus 8 bits every 125 μsec, or 64 kbps.

At the other end of the call, an analog signal is recreated from the quantized samples by playing them out (and smoothing them) over time. It will not be exactly the same as the original analog signal, even though we sampled at the Nyquist rate, because the samples were quantized. To reduce the error due to quantization, the quantization levels are unevenly spaced. A logarithmic scale is used that gives relatively more bits to smaller signal amplitudes and relatively fewer bits to large signal amplitudes. In this way the error is proportional to the signal amplitude.

Two versions of quantization are widely used: **μ-law**, used in North America and Japan, and **A-law**, used in Europe and the rest of the world. Both versions are specified in standard ITU G.711. An equivalent way to think about this process is to imagine that the dynamic range of the signal (or the ratio between the largest and smallest possible values) is compressed before it is (evenly) quantized, and then expanded when the analog signal is recreated. For this reason it is called

companding. It is also possible to compress the samples after they are digitized so that they require much less than 64 kbps. However, we will leave this topic for when we explore audio applications such as voice over IP.

Time Division Multiplexing

TDM based on PCM is used to carry multiple voice calls over trunks by sending a sample from each call every 125 μsec. When digital transmission began emerging as a feasible technology, ITU (then called CCITT) was unable to reach agreement on an international standard for PCM. Consequently, a variety of incompatible schemes are now in use in different countries around the world.

The method used in North America and Japan is the **T1** carrier, depicted in Fig. 2-37. (Technically speaking, the format is called DS1 and the carrier is called T1, but following widespread industry tradition, we will not make that subtle distinction here.) The T1 carrier consists of 24 voice channels multiplexed together. Each of the 24 channels, in turn, gets to insert 8 bits into the output stream.

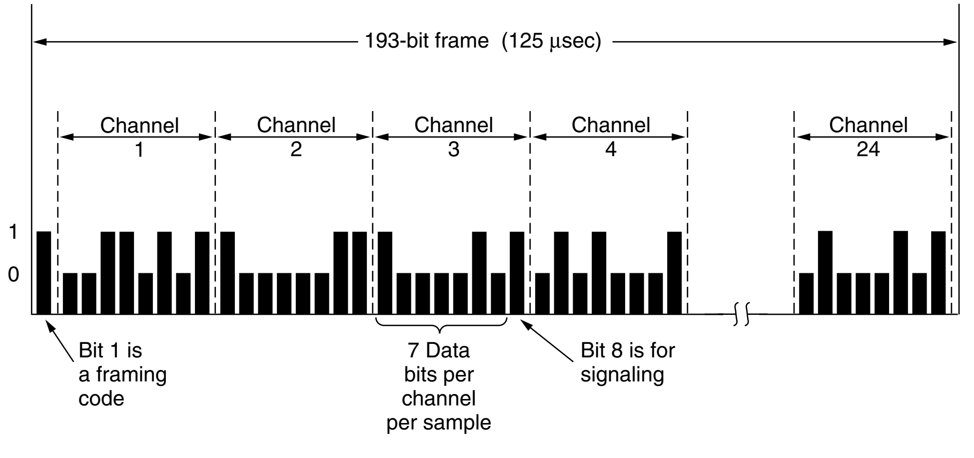


Figure 2-37. The T1 carrier (1.544 Mbps).

A frame consists of $24 \times 8 = 192$ bits plus one extra bit for control purposes, yielding 193 bits every 125 μsec. This gives a gross data rate of 1.544 Mbps, of which 8 kbps is for signaling. The 193rd bit is used for frame synchronization and signaling. In one variation, the 193rd bit is used across a group of 24 frames called an **extended superframe**. Six of the bits, in the 4th, 8th, 12th, 16th, 20th, and 24th positions, take on the alternating pattern 001011 . . . . Normally, the receiver keeps checking for this pattern to make sure that it has not lost synchronization. Six more bits are used to send an error check code to help the receiver confirm that it is synchronized. If it does get out of sync, the receiver can scan for the pattern and validate the error check code to get resynchronized. The remaining 12 bits are used for control information for operating and maintaining the network, such as performance reporting from the remote end.

The T1 format has several variations. The earlier versions sent signaling information **in-band**, meaning in the same channel as the data, by using some of the data bits. This design is one form of **channel-associated signaling**, because each channel has its own private signaling subchannel. In one arrangement, the least significant bit out of an 8-bit sample on each channel is used in every sixth frame. It has the colorful name of **robbed-bit signaling**. The idea is that a few stolen bits will not matter for voice calls. No one will hear the difference.

For data, however, it is another story. Delivering the wrong bits is unhelpful, to say the least. If older versions of T1 are used to carry data, only 7 of 8 bits, or 56 kbps can be used in each of the 24 channels. Instead, newer versions of T1 provide clear channels in which all of the bits may be used to send data. Clear channels are what businesses who lease a T1 line want when they send data across the telephone network in place of voice samples. Signaling for any voice calls is then handled **out-of-band**, meaning in a separate channel from the data. Often, the signaling is done with **common-channel signaling** in which there is a shared signaling channel. One of the 24 channels may be used for this purpose.

Outside North America and Japan, the 2.048-Mbps **E1** carrier is used instead of T1. This carrier has 32 8-bit data samples packed into the basic 125-μsec frame. Thirty of the channels are used for information and up to two are used for signaling. Each group of four frames provides 64 signaling bits, half of which are used for signaling (whether channel-associated or common-channel) and half of which are used for frame synchronization or are reserved for each country to use as it wishes.

Time division multiplexing allows multiple T1 carriers to be multiplexed into higher-order carriers. Figure 2-38 shows how this can be done. At the left we see four T1 channels being multiplexed into one T2 channel. The multiplexing at T2 and above is done bit for bit, rather than byte for byte with the 24 voice channels that make up a T1 frame. Four T1 streams at 1.544 Mbps should generate 6.176 Mbps, but T2 is actually 6.312 Mbps. The extra bits are used for framing and recovery in case the carrier slips. T1 and T3 are widely used by customers, whereas T2 and T4 are only used within the telephone system itself, so they are not well known.

At the next level, seven T2 streams are combined bitwise to form a T3 stream. Then six T3 streams are joined to form a T4 stream. At each step a small amount of overhead is added for framing and recovery in case the synchronization between sender and receiver is lost.

Just as there is little agreement on the basic carrier between the United States and the rest of the world, there is equally little agreement on how it is to be multiplexed into higher-bandwidth carriers. The U.S. scheme of stepping up by 4, 7, and 6 did not strike everyone else as the way to go, so the ITU standard calls for multiplexing four streams into one stream at each level. Also, the framing and

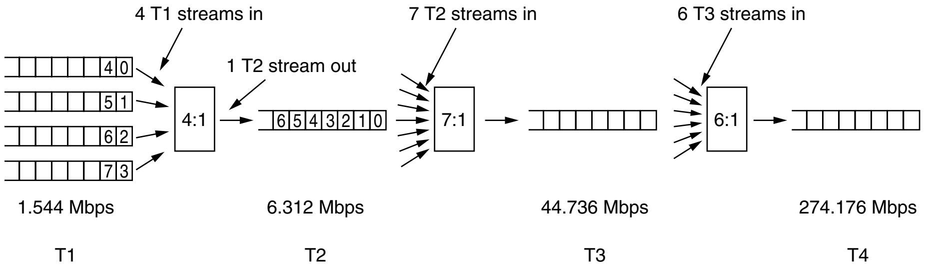


Figure 2-38. Multiplexing T1 streams into higher carriers.

recovery data are different in the U.S. and ITU standards. The ITU hierarchy for 32, 128, 512, 2048, and 8192 channels runs at speeds of 2.048, 8.848, 34.304, 139.264, and 565.148 Mbps.

SONET/SDH

In the early days of fiber optics, every telephone company had its own proprietary optical TDM system. After AT&T was broken up in 1984, local telephone companies had to connect to multiple long-distance carriers, all with different optical TDM systems, so the need for standardization became obvious. In 1985, Bellcore, the RBOC's research arm, began working on a standard, called **SONET** (**Synchronous Optical NETwork**).

Later, ITU joined the effort, which resulted in a SONET standard and a set of parallel ITU recommendations (G.707, G.708, and G.709) in 1989. The ITU recommendations are called **SDH** (**Synchronous Digital Hierarchy**) but differ from SONET only in minor ways. Virtually all the long-distance telephone traffic in the United States, and much of it elsewhere, now uses trunks running SONET in the physical layer. For additional information about SONET, see Bellamy (2000), Goralski (2002), and Shepard (2001).

The SONET design had four major goals. First and foremost, SONET had to make it possible for different carriers to interwork. Achieving this goal required defining a common signaling standard with respect to wavelength, timing, framing structure, and other issues.

Second, some means was needed to unify the U.S., European, and Japanese digital systems, all of which were based on 64-kbps PCM channels but combined them in different (and incompatible) ways.

Third, SONET had to provide a way to multiplex multiple digital channels. At the time SONET was devised, the highest-speed digital carrier actually used widely in the United States was T3, at 44.736 Mbps. T4 was defined, but not used