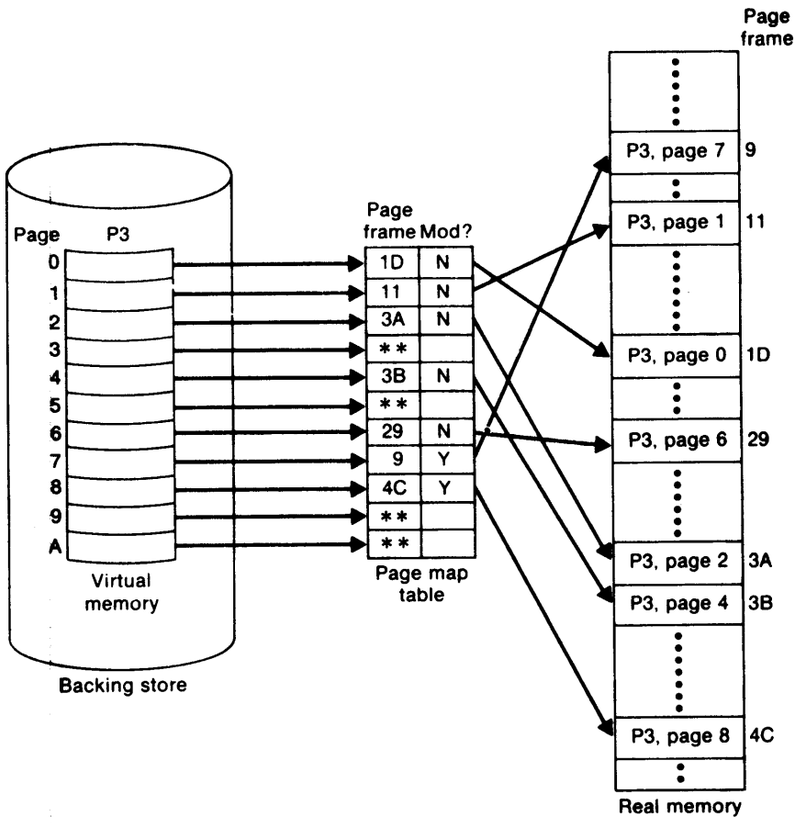


Figure 6.19 Virtual-to-real mapping using a page map table.

time. For example, it is clearly advantageous to allow other programs to use the CPU while paging I/O is being performed. The management of this multiprogramming requires that the interrupt system be available. Therefore, the page fault interrupt handler first determines what actions need to be performed and saves the status information from the interrupted process. The interrupt handler than enables the interrupt system during the remainder of the processing.

More specifically, the interrupt handler selects a page frame to receive the required page and marks this frame as *committed* so that it will not be selected again because of a subsequent page fault. If a page is to be removed, the PMT for the process that owns that page is updated to reflect its removal. This prevents that process from attempting to reference the page while it is being removed. The interrupt handler saves the status information from the program-interrupt work area, placing it in some location associated with the process so that this

```
procedure DAT {implemented in hardware}

    decompose virtual address into (page number, offset)
    find entry in PMT for this page
    if the page is currently in memory then
       begin
          combine (page frame address, offset) to form real address
          if this is a "store" instruction then
             mark PMT entry to indicate page modified
       end {if page is in memory}
    else
          generate page fault interrupt
```

(a)

```
procedure PAGEFAULT {implemented as part of the operating system}

    save process status from interrupt work area
    mark process as Blocked
    if there is an empty page frame then
       begin
          select an empty page frame
          mark the selected page frame table entry as committed
          enable all interrupts using LPS
       end {if empty page frame}
    else
       begin
          select page to be removed
          mark the selected page frame table entry as committed
          update PMT to reflect the removal of the page
          enable all interrupts using LPS
          if the selected page has been modified then
             begin
                issue I/O request to rewrite page to backing store
                wait for completion of the write operation
             end {if modified}
       end {if no empty page frame}
    issue I/O request to read page into the selected page frame
    wait for completion of the read operation
    update PMT and page frame table
    mark process as Ready
    restore status of user process that caused the page fault
```

(b)

Figure 6.20 Algorithms for dynamic address translation and page fault interrupt processing.

information will not be destroyed by a subsequent program interrupt. Then the interrupt handler turns on the interrupt system by loading a status word that is the same as the current SW, except that all MASK bits are set to 1. The remainder of the interrupt-processing routine functions in much the same way as a user process: it makes SVC requests to initiate I/O operations and to wait for the results. After the completion of the paging operation, the interrupt handler uses the saved status information to return control to the instruction that caused the page fault. The dynamic address translation for this instruction will then be repeated.

The algorithm description in Fig. 6.20(b) leaves several important questions unanswered. The most obvious of these is which page to select for removal. Some systems keep records of when each page in memory was last referenced and replace the page that has been unused for the longest time. This is called the *least recently used* (LRU) method. Since the overhead for this kind of record keeping can be high, simpler approximations to LRU are often used. Other systems attempt to determine the set of pages that are frequently used by the process in question (the so-called *working set* of the process). These systems attempt to replace pages in such a way that each process always has its working set in memory. Discussions and evaluations of various page replacement strategies can be found in Tanenbaum (1992) and Deitel (1990).

Another unanswered question concerns the implementation of the page tables themselves. One possible solution is to implement these tables as arrays in central memory. A register is set by the operating system to point to the beginning of the PMT for the currently executing process. This method can be very inefficient because it requires an extra memory access for each address translation. Some systems, however, use such a technique in combination with a high-speed buffer to improve average access time. Another possibility is to implement the page map tables in a special high-speed associative memory. This is very efficient, but may be too expensive for systems with large real memories. Further discussions of these and other PMT implementation techniques can be found in Tanenbaum (1992).

Demand-paging systems avoid most of the wasted memory due to fragmentation that is often associated with partitioning schemes. They also save memory in other ways. For example, parts of a program that are not used during a particular execution need not be loaded. However, demand-paging systems are vulnerable to other serious problems. For example, suppose that referencing a word in central memory requires 1 μsec, and that fetching a page from the backing store requires an average of 10 msec (10,000 μsec). Suppose also that on the average, considering all jobs in the system, only 1 out of 100 virtual memory references causes a page fault. Even with this apparently low page fault rate, the system will not perform well. For every 100 memory references (requiring 100 μsec), the system will spend 10,000 μsec fetching pages

from the backing store. Thus the computing system will spend approximately 99 percent of its time swapping pages, and only 1 percent of its time doing useful work. This total collapse of service because of a high paging rate is known as *thrashing*.

To avoid thrashing in the situation just described, it is necessary for the page fault rate to be much lower (perhaps on the order of one fault for every 10,000 memory references). At first glance, this might seem to make demand paging useless. It appears that all of a program's pages would need to be in memory to achieve acceptable performance. However, this is not necessarily the case. Because of a property called *locality of reference*, which can be observed in most real programs, memory references are not randomly distributed through a program's virtual address space. Instead, memory references tend to be clustered together in the address space, as illustrated in Fig. 6.21(a). This clustering is due to common program characteristics such as sequential instruction execution, compact coding of loops, sequential processing of data structures, and so on.

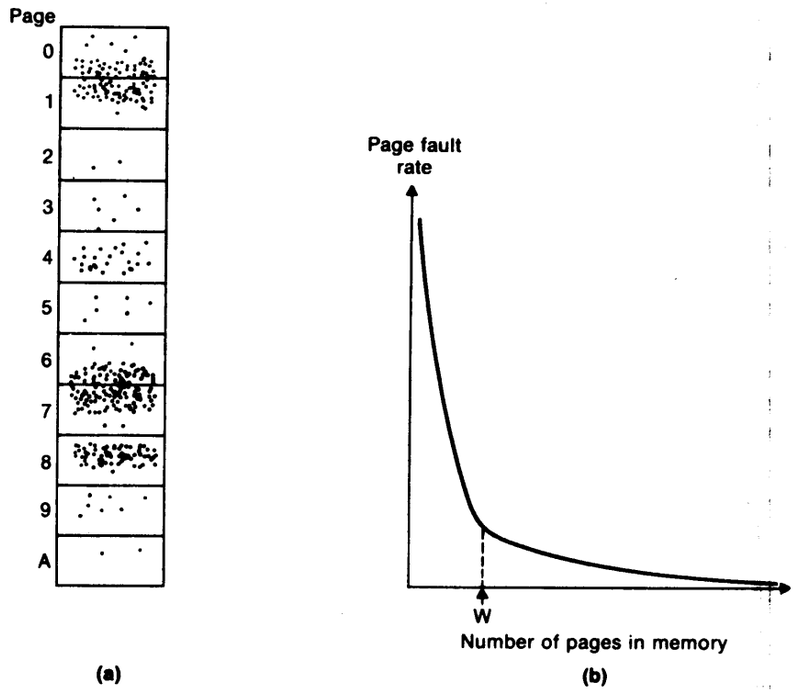


Figure 6.21 (a) Localized memory references. (b) Effect of localized references on page fault rate.

Because of locality of reference, it is possible to achieve an acceptably low page fault rate without keeping all of a program's address space in real memory. Figure 6.21(b) shows the page fault rate for a hypothetical program as a function of the number of the program's pages kept in memory. The scaling of this curve varies markedly from one program to another; however, the general shape is typical of many real programs. Often, as in this example, there is a critical point W. If fewer than W pages are in memory, thrashing will occur. If W pages or more are in memory, performance will be satisfactory. This critical point W is the size of the program's working set of pages that was mentioned earlier. For further discussion of the issues of working-set size and thrashing, see Tanenbaum (1992) and Deitel (1990).

Demand paging provides yet another example of delayed binding: the association of a virtual-memory address with a real-memory address is not made until the memory reference is performed. This delayed binding requires more overhead (for dynamic address translation, page fetching, etc.). However, it can provide more convenience for the programmer and more effective use of real memory. You may want to compare these observations with those made in the previous examples of delayed binding (Sections 3.4.2, 5.3.3, and 5.4.2).

In this section we have described an implementation of virtual memory using demand paging. A different type of virtual memory can be implemented using a technique called *segmentation.* In a segmented virtual-memory system, an address consists of a segment number and an offset within the segment being addressed. The concepts of mapping and dynamic address translation are similar to those we have discussed. However, in most systems segments may be of any length (as opposed to pages, which are usually of a fixed length for the entire system). Also, segments usually correspond to logical program units such as procedures or data areas (as opposed to pages, which are arbitrary divisions of the address space). This makes it possible to associate protection attributes such as *read only* or *execute only* with certain segments. It is also possible for segments to be shared between different user jobs. Segmentation is often combined with demand paging. This combination requires a two-level mapping and address-translation procedure. For further information about segmentation and its implementation, see Deitel (1990) and Tanenbaum (1992).

6.3 MACHINE-INDEPENDENT OPERATING SYSTEM FEATURES

In this section we briefly describe several common functions of operating systems that are not directly related to the architecture of the machine on which the system runs. These features tend to be implemented at a higher level—that

is, further removed from the machine level—than the features we have discussed so far. For this reason, such topics are not as fundamental to the basic purpose of an operating system as are the hardware-support topics discussed in the last section. Because of space limitations, these subjects are not discussed in as much detail as were the machine-dependent features. References are provided for readers who want to learn more about the topics introduced here.

In Section 6.2.3 we described a technique that can be used to manage I/O operations. Section 6.3.1 examines a related topic at a higher level: the management and processing of logical files. Similarly, Section 6.3.2 discusses the problem of job scheduling, which selects user jobs as candidates for the lower-level process scheduling discussed previously.

Section 6.3.3 discusses the general subject of resource allocation by an operating system and describes some of the problems that may occur.

6.3.1 File Processing

In this section we introduce some of the functions performed by a typical operating system in managing and processing files. On most systems, it is possible for user programs to request I/O by using mechanisms like those described in Section 6.2.3—for example, constructing channel programs and making SVC requests to execute them. However, this is generally inconvenient. The programmer is required to be familiar with the details of the I/O command codes and formats. The user program must know the correct physical device address. In the case of a direct-access device such as a disk, the program also must know the actual address of the desired record on the device. In addition, the program must take care of details such as waiting for I/O completion, and the buffering and blocking functions described later in this section.

The file-management function of an operating system is an intermediate stage between the user program and the I/O supervisor. This function is illustrated in Fig. 6.22. The user program makes requests, such as "Read the next record from file F," at a logical level, using file names, keys, etc. The file-management routine, which is sometimes called an *access method,* translates these logical requests into physical I/O requests, and passes the requests to the I/O supervisor. The I/O supervisor then proceeds as described in Section 6.2.3 to manage the physical I/O operations.

To convert the program's logical requests into physical I/O requests, the file manager must have information about the location and structure of the file. It obtains such information from data structures we call the *catalog* and the *file information tables.* The actual terms used for these structures, as well as their formats and contents, vary considerably from one operating system to another. The catalog relates logical file names to their physical locations and may give some basic information about the files. The file information table for

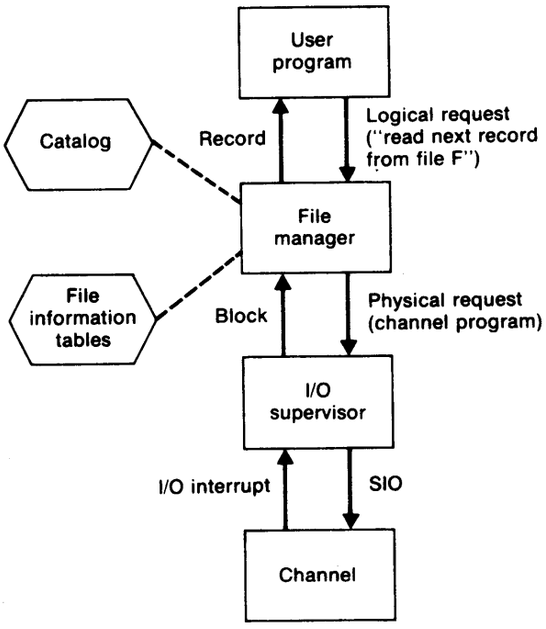


Figure 6.22 I/O using a file manager routine.

a file gives additional information such as file organization, record length and format, and indexing technique, if any. To begin the processing of a file, the file manager searches the catalog and locates the appropriate file information table. The file manager may also create buffer areas to receive the blocks being read or written. This initialization procedure is known as *opening* the file. After the processing of the file is completed, the buffers and any other work areas and pointers are deleted. This procedure is called *closing* the file.

One of the most important functions of the file manager is the automatic performance of *blocking* and *buffering* operations on files being read or written. Figure 6.23 illustrates these operations on a sequential input file. We assume the user program starts reading records at the beginning of the file and reads each record in sequence until the end. The file logically consists of records that are 1024 bytes long; however, the file is physically written in 8192-byte blocks, with each block containing 8 logical records. This sort of blocking of records is commonly done with certain types of storage devices to save processing time and storage space.

Figure 6.23(a) shows the situation after the file has been opened and the user program has made its first read-record request. The file manager has issued an I/O request to read the first block of the file into buffer B1. The file

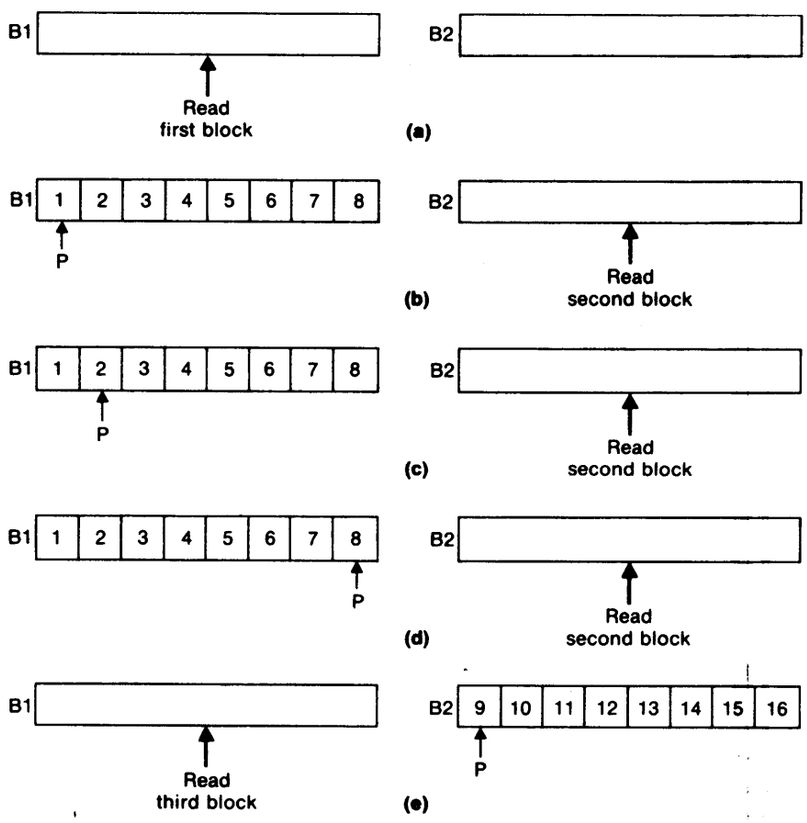


Figure 6.23 Blocking and buffering of a sequential file.

manager must wait for the completion of this I/O operation before it can return the requested record to the user. In Fig. 6.23(b), the first block has been read. This block, containing logical records 1 through 8, is present in buffer B1. The file manager can now return the requested record to the user program. In this case, the requested record is returned by setting a pointer P to the first logical record. The file manager also issues a second physical I/O request to read the second block of the file into buffer B2.

The next time the user program makes a read-record request, it is not necessary to wait for any physical I/O activity. The file manager simply advances the pointer P to logical record 2, and returns to the user. This operation is illustrated in Fig. 6.23(c). Note that the physical I/O operation that reads the second block into buffer B2 is still in progress. The same process continues for the rest of the logical records in the first block [see Fig. 6.23(d)].

If the user program makes its ninth read-record request before the completion of the I/O operation for block 2, the file manager must again cause the program to wait. After the second block has been read, the pointer P is switched to the first record in buffer B2. The file manager then issues another I/O request to read the third block of the file into buffer B1, and the process continues as just described. Note that the use of two buffer areas allows overlap of the internal processing of one block with the reading of the next. This technique, often called *double buffering*, is widely used for input and output of sequential files.

The user program in the previous example simply makes a series of read-record requests. It is unaware of the buffering operations and of the details of the physical I/O requests being performed. Compare this with the program in Fig. 6.11, which performs a similar buffering function by dealing directly with the I/O supervisor. Clearly, the use of the file manager makes the user program much simpler and easier to write, and therefore less error-prone. It also avoids the duplication of similar code in a large number of programs.

File-management routines also perform many other functions, such as the allocation of space on external storage devices and the implementation of rules governing file access and use. For further discussions of such topics, see Deitel (1990) and Tanenbaum (1992).

6.3.2 Job Scheduling

Job scheduling is the task of selecting the next user job to begin execution. In a single-job system, the job scheduler completely specifies the order of job execution. In a multiprogramming system, the job scheduler specifies the order in which jobs enter the set of tasks that are being executed concurrently.

Figure 6.24(a) illustrates a typical two-level scheduling scheme for a multiprogramming system. Jobs submitted to the system become part of an *input queue*; a *job scheduler* selects jobs from this workload. The jobs selected become *active*, which means they begin to participate in the process-scheduling operation described in Section 6.2.2. This two-stage procedure is used to limit the *multiprogramming level*, which is the number of user jobs sharing the CPU and the other system resources. Such a limitation is necessary in a multiprogramming system to maintain efficient operation. If the system attempts to run too many jobs concurrently, the overhead of resource management becomes too large, and the amount of resources available to each job becomes too small. As a result, system performance is degraded.

In the scheme just described, the job scheduler is used as a tool to maintain a desirable level of multiprogramming. However, this ideal multiprogramming level may vary according to the jobs being executed. Consider, for example, a system that uses demand-paged memory management. The number of

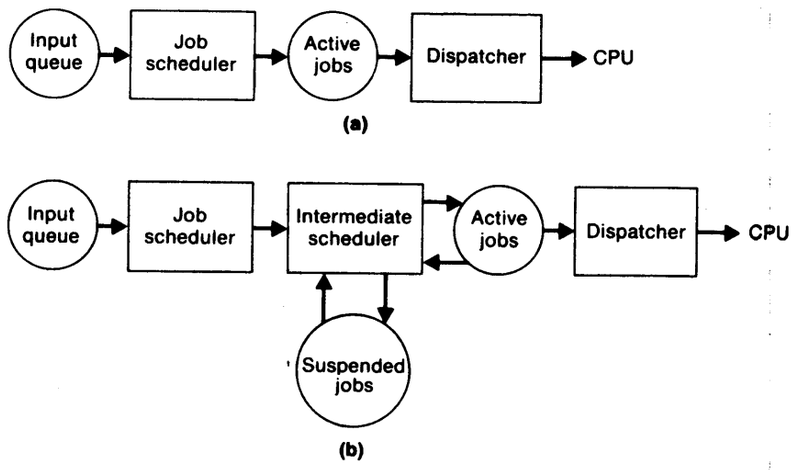


Figure 6.24 (a) Two-level scheduling system and (b) three-level scheduling system.

user jobs that can share the real memory is essentially unlimited. Each job can potentially be executed with as little as one page in memory. However, thrashing occurs when a job does not have a certain critical number of pages in memory, and the performance of the overall system suffers. Unfortunately, the number of pages a job requires to prevent thrashing is difficult to predict. In addition, this critical number of pages may change considerably during the execution of the program, so the desired level of multiprogramming may change during the operation of the system.

The multiprogramming level can be increased easily enough by simply invoking the job scheduler, assuming that the input queue is not empty. It is more difficult to decrease the level of multiprogramming, which would be done, for example, to stop a system from thrashing. Figure 6.24(b) shows a three-level scheduling procedure that is commonly used to accomplish this decrease. The job scheduler and the process scheduler (i.e., the dispatcher) operate as before. However, there is also an *intermediate-level scheduler* that monitors system performance and adjusts the multiprogramming level as needed. If the multiprogramming level is too high, the intermediate scheduler lowers it by *suspending* or *rolling out* one or more active jobs. If the multiprogramming level is too low, the intermediate scheduler resumes the execution of a suspended job or calls on the job scheduler for a new job to be made active. Such intermediate schedulers can also be used to adjust the dispatching priority of active jobs, based on observation of the jobs during execution.

The overall scheduling system is usually based on a system of priorities that are designed to help meet desired goals. For example, one goal might be to achieve the maximum system *throughput*—that is, to perform the most computing work in the shortest time. Clearly, achieving this goal is based on making effective use of overall system resources. Another common goal is to achieve the lowest average *turnaround time,* which is the time between the submission of a job by a user and the completion of that job. A related goal for a time-sharing system is to minimize expected *response time,* which is the length of time between entering a command and beginning to receive a response from the system.

There are many other possible scheduling goals for a computing system. For example, we might want to provide a guaranteed level of service by limiting the maximum possible turnaround time or response time. Another alternative is to be equitable by attempting to provide the same level of service for all. On the other hand, it may be desirable to give certain jobs priority for external reasons such as meeting deadlines or providing good service to important or influential users. On some systems it is even possible for users to get higher priority by paying higher rates for service, in which case the overall scheduling goal of the system might be to make the most money.

The first two goals mentioned above—high throughput and low average turnaround time or response time—are commonly accepted as desirable system characteristics. Unfortunately, these two goals are often incompatible. Consider, for example, a time-sharing system with a large number of terminals. We might choose to provide better response time by switching control more rapidly among the active user terminals. This could be accomplished by giving each process a shorter time-slice when it is dispatched. However, the use of shorter time-slices would mean a higher frequency of context switching operations, and would require the operating system to make more frequent decisions about the allocation of the CPU and other resources. This means the operating system overhead would be higher and correspondingly less time would be available for user jobs, so the overall system throughput would decline.

On the other hand, consider a batch processing system that runs one job at a time. The execution of two jobs on such a system is illustrated in Fig. 6.25(a). Note the periods of CPU idle time, represented by gaps in the horizontal lines for Jobs 1 and 2. If both jobs are submitted at time 0, then the turnaround time for Job 1 (T_1) is 2 minutes, and the turnaround time for Job 2 (T_2) is 5 minutes. The average turnaround time, T_{avg}, is 3.5 minutes.

Now consider a multiprogramming system that runs two jobs concurrently, as illustrated in Fig. 6.25(b). Note that the two concurrent jobs share the CPU, so there is less overall idle time; this is the same phenomenon we studied in Fig. 6.15. Because there is less idle time, the two jobs are completed in less total time: 4.5 minutes instead of 5 minutes. This means the system throughput has been improved: we have done the same amount of work in

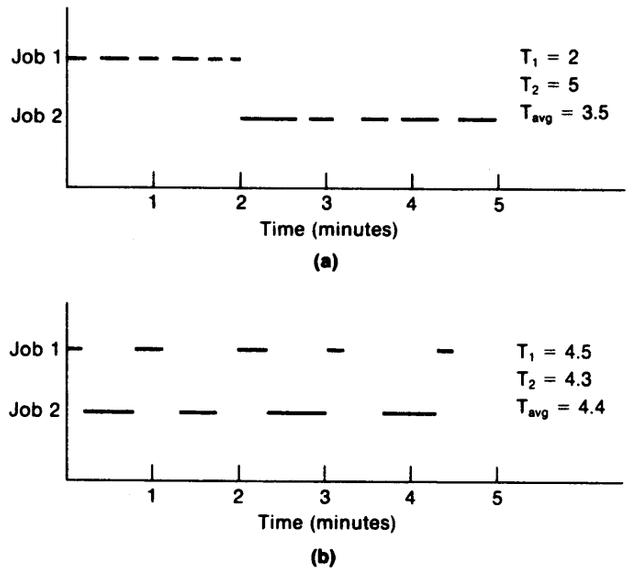


Figure 6.25 Comparison of turnaround time and throughput for (a) a single-job system and (b) a multiprogramming system.

less time. However, the average turnaround time has become worse: 4.4 minutes instead of 3.5 minutes.

Two common job-scheduling policies are *first come-first served* (FCFS) and *shortest job first* (SJF). FCFS tends to treat all jobs equally, so it minimizes the range of turnaround times. SJF provides a lower average turnaround time because it runs short jobs much more quickly; however, long jobs may be forced to wait a long time for service. For examples of these characteristics and discussions of other scheduling policies, see Deitel (1990).

6.3.3 Resource Allocation

In Section 6.2 we discussed how an operating system might control the use of resources such as central memory, I/O channels, and the CPU. These resources are needed by all user jobs, and their allocation is handled automatically by the system. In this section, we describe a more general resource-allocation function provided by many operating systems. Such a facility can be used to control the allocation of user-defined resources such as files and data structures.

The need for a general resource-allocation function is illustrated by the two programs in Fig. 6.26(a). Both these programs utilize a sequential stack that is defined by some other program. The external variable STACK indicates the

```
 1     P1      START     0
 2             EXTREF    STACK,TOP
 3             LDS       #3          REGISTER S = CONSTANT 3
               .
               .
               .
24             +LDX      TOP         GET POINTER TO TOP OF STACK
25              ADDR     S,X         INCREMENT POINTER
26             +STA      STACK,X     ADD NEW ITEM TO STACK
27             +STX      TOP         STORE NEW TOP POINTER
               .
               .
               .
48             END
```

```
 1     P2      START     0
 2             EXTREF    STACK, TOP
 3             LDS       #3          REGISTER S = CONSTANT 3
               .
               .
               .
37             +LDX      TOP         GET POINTER TO TOP OF STACK
38             +LDA      STACK,X     GET TOP ITEM FROM STACK
39              SUBR     S,X         DECREMENT POINTER
40             +STX      TOP         STORE NEW TOP POINTER
               .
               .
               .
75             END
```

(a)

Figure 6.26 Control of resources using operating system service requests.

base address of the stack; TOP contains the relative location of the item currently on top of the stack. We assume that external references to the variables STACK and TOP are handled by linking methods like those discussed in Chapter 3. Program P1 adds items to the stack by incrementing the previous value of TOP by 3, storing a new item from register A on the top of the stack, and then saving the new value of TOP (lines 24–27). Program P2 removes items by loading the value from the top of the stack into register A, and then subtracting 3 from the value of TOP (lines 37–40). For simplicity, we have not shown the code needed to handle stack overflow and underflow.

```
 1     P1       START     0
 2              EXTREF    STACK,TOP
 3              LDS       #3            REGISTER S = CONSTANT 3
                .
                .
                .
22              LDT       =SNAME        SET POINTER TO RESOURCE NAME
23              SVC       3             REQUEST RESOURCE
24              +LDX      TOP           GET POINTER TO TOP OF STACK
25              ADDR      S,X           INCREMENT POINTER
26              +STA      STACK,X       ADD NEW ITEM TO STACK
27              +STX      TOP           STORE NEW TOP POINTER
28              SVC       4             RELEASE RESOURCE
                .
                .
                .
47     SNAME    BYTE      C'STACK1'
48              END
```

```
 1     P2       START     0
 2              EXTREF    STACK,TOP
 3              LDS       #3            REGISTER S = CONSTANT 3
                .
                .
                .
35              LDT       =STKNM        SET POINTER TO RESOURCE NAME
36              SVC       3             REQUEST RESOURCE
37              +LDX      TOP           GET POINTER TO TOP OF STACK
38              +LDA      STACK,X       GET TOP ITEM FROM STACK
39              SUBR      S,X           DECREMENT POINTER
40              +STX      TOP           STORE NEW TOP POINTER
41              SVC       4             RELEASE RESOURCE
                .
                .
                .
74     STKNM    BYTE      C'STACK1'
75              END
```

(b)

Figure 6.26 (*cont'd*)

If processes P1 and P2 are executed concurrently, they may or may not work properly. For example, suppose the present value of TOP is 12. If P1 executes its instructions numbered 24–27, it will add a new item in bytes 15–17 of the stack, and the new value of TOP will be 15. If P2 then executes its

instructions 37–40, it will remove the item just added by P1, resetting the value of TOP to 12. This represents a correct functioning of P1 and P2: the two processes perform their intended operations on the stack without interfering with each other. Another correct sequence would occur if P2 executed lines 37–40, and then P1 executed lines 24–27.

On the other hand, suppose that P1 has just executed line 24 when its current time-slice expires. The resulting timer interrupt causes all register values to be saved; the saved value for register X is 12. Suppose now that the dispatcher transfers control to P2, which executes lines 37–40. These instructions will cause P2 to remove the item from bytes 12–14 of the stack because the value of TOP has not yet been updated by P1; P2 will then set TOP to 9. When P1 regains control of the CPU, its register X will still contain the value 12. Thus lines 25–27 will add the new item in bytes 15–17 of the stack, setting TOP to 15.

The sequence of events just described has resulted in an incorrect operation of P1 and P2. The item that was removed by P2 is still logically a part of the stack, and the stack appears to contain one more item than it should. Several other sequences of execution also yield incorrect results. Similar problems may occur whenever two concurrent processes attempt to update the same file or data structure.

Problems of this sort can be prevented by granting P1 or P2 exclusive control of the stack during the time it takes to perform the updating operations. Figure 6.26(b) illustrates a common type of solution using operating system service requests. P1 requests exclusive control of the stack by executing an SVC 3 instruction. The resource being requested is specified by register T, which points to the (user-defined) logical name that has been assigned to the stack. After adding the new item to the stack and updating TOP, P1 releases control of the stack by executing an SVC 4. P2 performs a similar sequence of request and release operations.

The operating system responds to a request for control of a resource by checking whether or not that resource is currently assigned to some other process. If the requested resource is free, the operating system returns control to the requesting process. If the resource is busy, the system places the requesting process in the blocked state until the resource becomes available. For example, suppose the resource STACK1 is currently free. If P1 requests this resource (line 23), the system will return control directly to P1. As in the previous discussion, suppose the time-slice for P1 expires after line 24 has just been executed. Control of the CPU then passes to P2; however, STACK1 remains assigned to P1. Thus P2 will be placed in the blocked state when it requests STACK1 at line 36. Eventually P1 will regain control and complete its operation on the stack. When P1 releases control of STACK1 (line 28), P2 will be assigned control of this resource and moved from the blocked state to the

```
 1      P3      START         0
                .
                .
 2              LDT           =R1             REQUEST RES1
 3              SVC           3
                .
                .
 4              LDT           =R2             REQUEST RES2
 5              SVC           3
                .
                .
 6              LDT           =R2             RELEASE RES2
 7              SVC           4
                .
                .
 8              LDT           =R1             RELEASE RES1
 9              SVC           4
                .
                .
10      R1      BYTE          C'RES1    '
11      R2      BYTE          C'RES2    '
12              END


 1      P4      START         0
                .
                .
 2              LDT           =R2             REQUEST RES2
 3              SVC           3
                .
                .
 4              LDT           =R1             REQUEST RES1
 5              SVC           3
                .
                .
 6              LDT           =R1             RELEASE RES1
 7              SVC           4
                .
                .
 8              LDT           =R2             RELEASE RES2
 9              SVC           4
                .
                .
10      R1      BYTE          C'RES1    '
11      R2      BYTE          C'RES2    '
12              END
```

Figure 6.27 Resource requests leading to potential deadlock.

ready state. It can then continue with its updating operation when it next receives control of the CPU. You should trace through this sequence of events carefully to see how the problems previously discussed are prevented by this scheme.

Unfortunately, the use of request and release operations can lead to other types of problems. Consider, for example, the programs in Fig. 6.27. P3 first requests control of resource RES1; later, it requests resource RES2. P4 utilizes the same two resources; however, it requests RES2 before RES1.

Suppose P3 requests, and receives, control of RES1, and that its time-slice expires before it can request RES2. P4 may then be dispatched. Suppose P4 requests, and receives, control of RES2. This sequence of events creates a situation in which neither P3 nor P4 can complete its execution. Eventually, P4 will reach its line 5 and request control of RES1; it will then be placed into the blocked state because RES1 is assigned to P3. Similarly, P3 will eventually reach its line 5 and request control of RES2; P3 will then be blocked because RES2 is assigned to P4. Neither process can acquire the resource it needs to continue, so neither process will ever release the resource needed by the other.

This situation is an example of a *deadlock:* a set of processes each of which is permanently blocked because of resources held by the others. Once a deadlock occurs, the only solution is to release some of the resources currently being held; this usually means canceling one or more of the jobs involved. There are a number of methods that can prevent deadlocks from occurring. For example, the system could require that a process request all its resources at the same time, or that it request them in a particular order (such as RES1 before RES2). Unfortunately, such methods may require that resources be tied up for longer than is really necessary, which can degrade the overall operation of the system. Discussions of methods for detecting and preventing deadlocks can be found in Singhal and Shivaratri (1994) and Tanenbaum (1992).

The problems we have discussed in this section are examples of the more general problems of *mutual exclusion* and *process synchronization.* Discussions of these problems, and techniques for their solution, can be found in Singhal and Shivaratri (1994) and Tanenbaum (1992).

6.4 OPERATING SYSTEM DESIGN OPTIONS

In this section we briefly describe some important concepts related to the design and structure of an operating system. Sections 6.4.1 and 6.4.2 discuss operating systems for multiprocessors and distributed systems, and describe some options for the division of tasks between the processors.

6.4.1 Multiprocessor Operating Systems

Our previous discussions of operating systems have primarily concerned machines with one central processing unit (CPU). In this section we briefly describe some design alternatives for multiprocessor operating systems.

An operating system for a machine with multiple processors must perform the same basic tasks we have discussed for single-processor systems. Of course, some of these functions may need to be modified. For example, the process scheduler may have more than one CPU to assign to user jobs, so more than one process might be in the running state at the same time. There are also a number of less obvious issues, many of which are related to the overall organization of the system.

Figure 6.28 illustrates two possible types of multiprocessor architecture. In a *loosely coupled* system, each processor has its own logical address space and its own memory (and possibly other resources). The processors can communicate with each other, but each can directly access only its own local memory. In a *tightly coupled* system, all processors share the same logical address space, and there is a common memory that can be accessed by all processors. These types

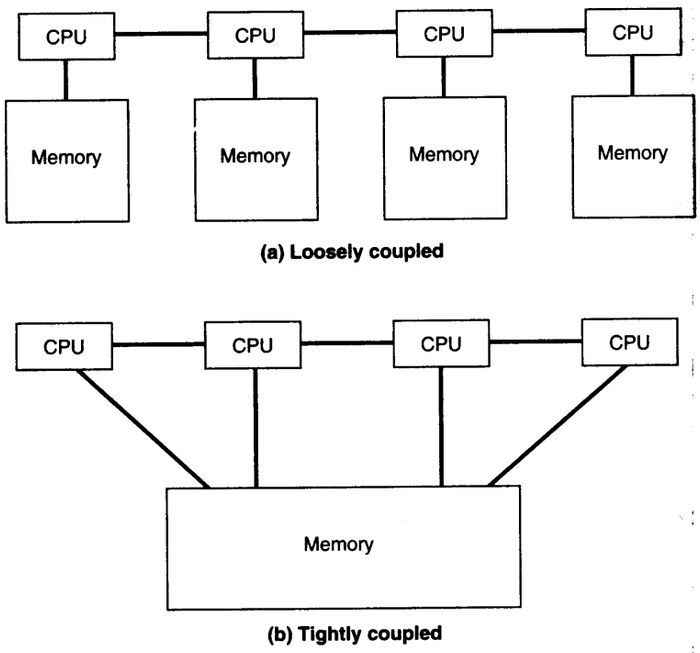


Figure 6.28 Types of multiprocessor architecture.

of multiprocessor organization are sometimes referred to as *distributed memory* systems and *shared memory* systems. However, it is possible to have an organization in which the memory is physically distributed but logically shared. (See, for example, the description of the Cray T3E architecture in Section 1.5.3.)

Figure 6.29 illustrates the three basic types of multiprocessor operating system. In a *separate supervisor* system, each processor has its own operating system. There are some common data structures that are used to synchronize communication between the processors. However, each processor acts largely as an independent system. Separate supervisor systems are relatively simple, and the failure of one processor need not affect the others. However, the independence between processors makes it difficult to perform parallel execution of a single user job.

In a *master-slave* system [Fig. 6.29(b)], one "master" processor performs all resource management and other operating system functions. This processor completely controls the activities of the "slave" processors, which execute user jobs. Essentially, the slave processors are treated as resources to be scheduled by the master. Thus it is possible to assign several slave processors to execute a user job in parallel. This type of operating system is rather similar to the single-processor systems we have discussed. Synchronization is easy, because all resources are controlled by the master processor.

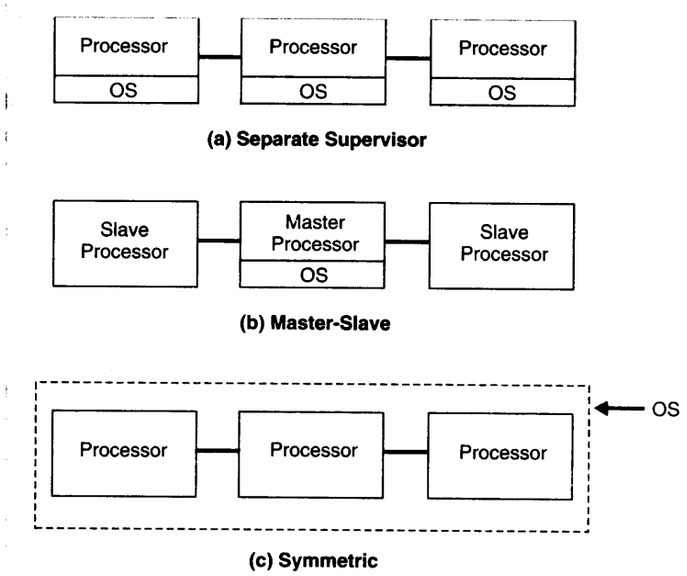


Figure 6.29 Types of multiprocessor operating systems.

The most significant problem with master-slave multiprocessing systems is the unbalanced use of resources. For example, the master processor might be overloaded by requests for operating system services, which could cause the slave processors to remain idle much of the time. In addition, any hardware failure in the master processor causes the entire system to stop functioning. Such problems can be avoided by allowing any processor to perform any function required by the operating system or by a user program. This approach, called *symmetric* processing, is illustrated in Fig. 6.29(c). Since all processors have the ability to perform the same sets of functions, the potential bottlenecks of a master-slave system are avoided. In addition, the failure of any one processor will not necessarily cause the entire system to fail. The other processors can continue to perform all the required functions.

In a symmetric multiprocessing system, different parts of the operating system can be executed simultaneously by different processors. Because of this, such a system may be significantly more complicated and more difficult to design than the other types of operating systems we have discussed. For example, all processors must have access to all the data structures used by the operating system. However, the processors deal with these data structures independently of one another, and problems can arise if two processors attempt to update the same structure at the same time (see Section 6.3.3). Symmetric multiprocessing systems must provide some mechanism for controlling access to critical operating system tables and data structures. The request and release operations described in Section 6.3.3 are not sufficient to handle this problem because two different processors might perform request operations simultaneously. The solution usually requires a special hardware feature that allows one processor to seize control of a critical resource, locking out all other processors in a single step. Discussions of such mechanisms, and further information about multiprocessor operating systems, can be found in Singhal and Shivaratri (1994).

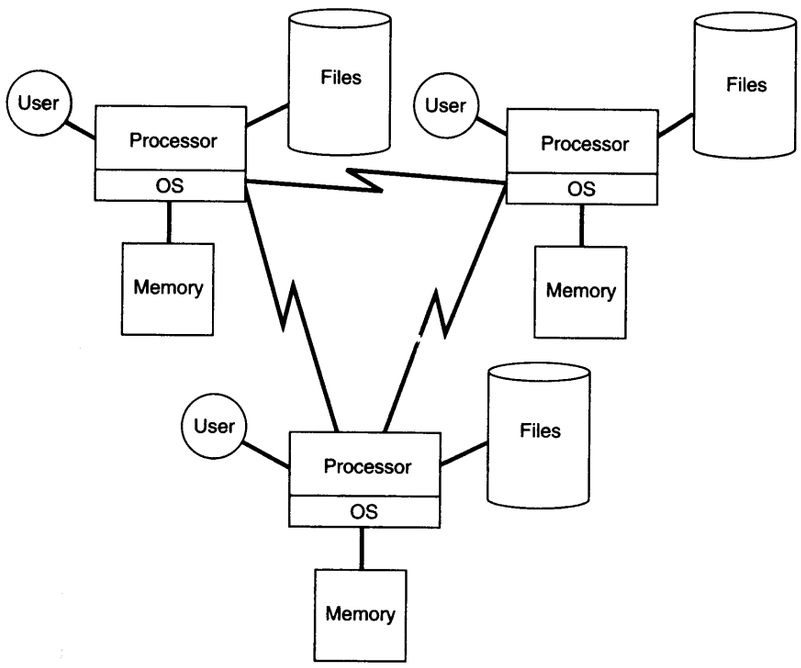
Section 6.5.4 describes an example of a symmetric multiprocessing operating system running on a machine with a tightly coupled architecture.

6.4.2 Distributed Operating Systems

Figure 6.30 illustrates two different approaches to operating systems for a network of computers. In Fig. 6.30(a), each machine on the network has its own independent operating system. These operating systems provide a communication interface that allows various types of interaction via the network. A user of such a system is aware of the existence of the network. He or she may login to remote machines, copy files from one machine to another, etc. This approach is often called a *network operating system.* Except for the network interface, such an operating system is quite similar to those found on a single-computer system.

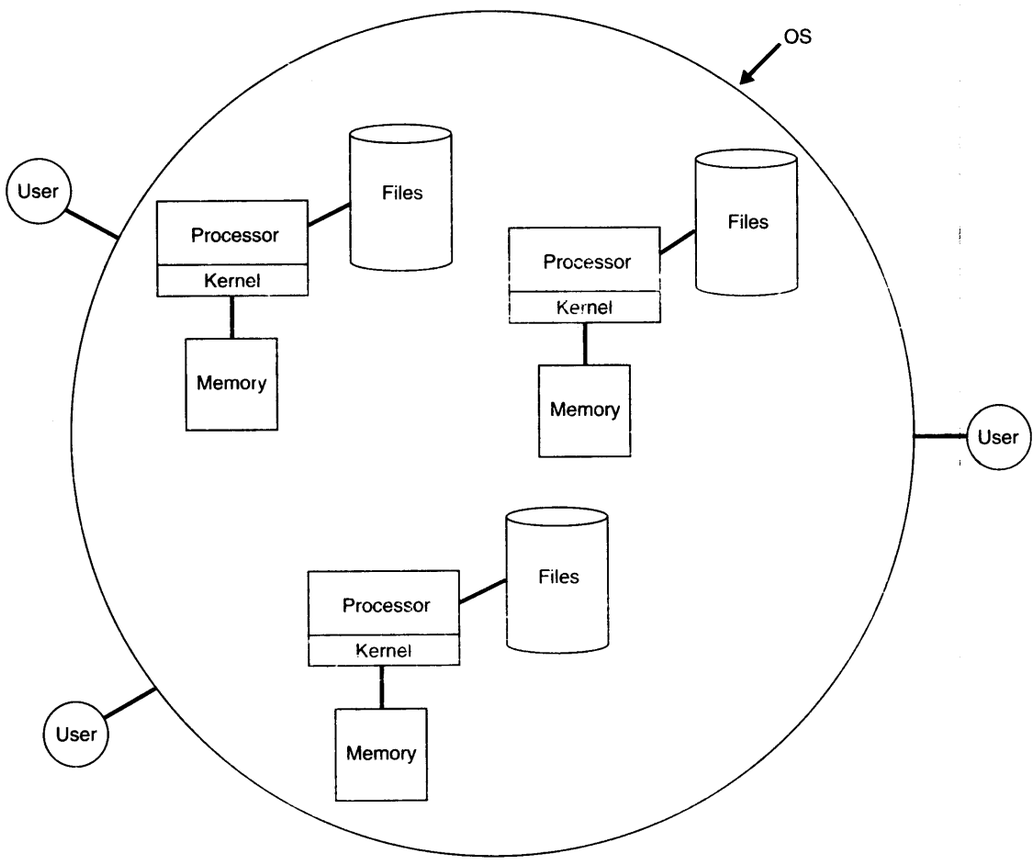
A *distributed operating system* [Fig. 6.30(b)] represents a more complex type of network organization. This kind of operating system manages hardware and software resources so that a user views the entire network as a single system. The user is not aware of which machine on the network is actually running a program or where the resources being used are actually located. In fact, many such systems allow programs to run on several processors at the same time.

Distributed operating systems have many advantages over traditional systems. The sharing of resources between computers is made easier—in fact, the user need not even be aware that sharing is taking place. A distributed system can provide improved performance by distributing the load between computers and executing parts of a task concurrently on several processors. Such a system can be more reliable, because the failure of one component need not affect the rest of the system. Likewise, adding additional processors or other resources can improve performance without requiring a major change in the system configuration.



(a) Network Operating Systems

Figure 6.30 Network and distributed operating systems.



(b) Distributed Operating Sytems

Figure 6.30 (*cont'd*)

The basic issues in a distributed operating system are similar to those in single-processor and multiprocessor systems. However, the solutions to problems such as process synchronization and the sharing of files and data are more complex. The machines controlled by a distributed operating system generally do not have access to a shared memory. Communication delays are unpredictable, and there is often no common time reference that can be used as a system clock. The machines on the network may be of different types, and they may have significantly different architectures. For example, the floating-point data formats and the choice of byte ordering may vary from one machine to another.

In order to give the appearance of a unified system, a distributed operating system must provide a consistent interface for users and their programs. For example, the mechanisms for requesting resources, communicating between processes, and making other types of operating system service requests must be the same on all machines in the system. This normally means that the same basic operating system kernel must be used on all machines. These design goals present a substantial challenge to system designers.

Further information about distributed operating systems can be found in Singhal and Shivaratri (1994). An example of one such system is described in Section 6.5.5 of this text.

6.4.3 Object-Oriented Operating Systems

This section describes how the concepts of object-oriented design and programming can be applied to operating systems. (If you are unfamiliar with object-oriented concepts, you may want to review Section 8.4 before proceeding.)

Figure 6.31 illustrates the general structure of an object-oriented operating system. Most of the system is implemented as a collection of *objects*. Objects

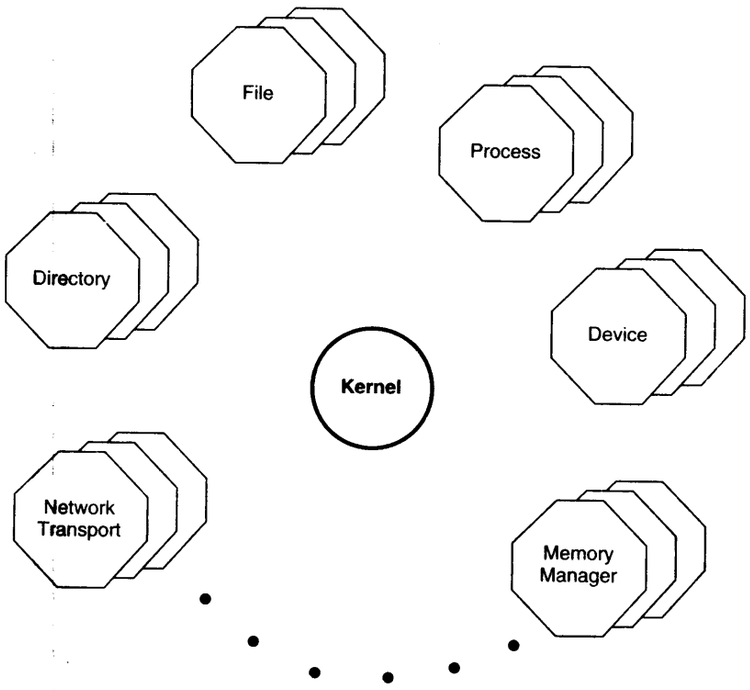


Figure 6.31 Object-oriented operating system structure.

belong to *classes* that designate some of the properties of the object. For example, there may be one class for file objects, one for processes, etc. Each object encapsulates a data structure and defines a set of operations on that data structure. For a file object, for example, typical operations are reading, writing, appending, and deleting. The operations defined for objects are called *methods*. When a user program (or a part of the operating system) needs to perform some operation on an object, it does so by *invoking* one of the methods defined for that object.

In a typical object-oriented operating system, this object-based model is implemented using processes called *servers* or *object managers*. When a process wants to invoke a method on an object, it sends a message to the server that manages that object. Servers also are responsible for creating new objects and deleting objects that are no longer needed. The *kernel* of the operating system is relatively small and simple. In a typical system, this kernel mediates communication (i.e., invoking methods on objects), provides some simple process scheduling functions, and does little else. Most of the functions that we normally consider part of the operating system, such as memory management and network interfaces, are performed by objects.

The object-oriented approach to operating system design has many advantages. The fundamental operations performed by all user processes and most operating system routines consist of invoking methods on objects. By invoking methods, processes can request operating system services, send messages to other processes, and perform remote procedure calls (running processes on other machines in a network). These three functions, which are implemented using different mechanisms in conventional operating systems, are all handled in the same way in an object-oriented system.

Object-oriented design provides a simple and natural approach to constructing distributed operating systems. From the user's point of view, invoking a method on an object at a remote machine on the network is exactly the same as invoking a method on a local object. Thus, distributed applications require no special handling. The implementation of the method (via an object manager or server) may be different on different machines. However, the details of the implementation are hidden from the invoking process—it sees only the interface by which the invocation is requested. Thus a distributed system that includes a variety of machines (possibly with widely differing architectures) presents no special problems.

The problem of providing security is also simplified. The actions that can be performed by a user process can be controlled by a system of *capabilities*. A capability gives a process the right to invoke a particular method on a specified object. These capabilities are validated by the kernel when it is involved in an invocation request. Since all access to objects is via invoking methods, this means that the part of the system that must be "trusted" is very small (i.e., the part of the kernel that mediates invocations).

Object-oriented operating systems are a relatively new development. However, many computer scientists believe that such systems will be widely used in the near future. Further discussions and examples of object-oriented operating systems may be found in Singhal and Shivaratri (1994) and Tanenbaum (1992). One example of such a system is briefly described in Section 6.5.5 of this text.

6.5 IMPLEMENTATION EXAMPLES

In this section we present brief descriptions of several real operating systems. These systems have been chosen to illustrate some of the variety of design and purpose in such software. As in our previous examples, we do not attempt a complete high-level description of any system. Instead, we focus on some of the more interesting or unusual features provided and give references for readers who want more information.

Section 6.5.1 discusses MS-DOS, which is a popular operating system for IBM-compatible personal computers. Section 6.5.2 describes Windows 95, a more sophisticated PC operating system that provides many more features than MS-DOS. Section 6.5.3 discusses SunOS, a popular version of the UNIX operating system that runs on SPARC, x86, and PowerPC systems.

The last two examples in this chapter are specialized systems that were designed for more complex architectures. Section 6.5.4 describes UNICOS/mk, Cray's multiprocessor operating system for the T3E.

6.5.1 MS-DOS

Version 1 of MS-DOS was written in 1981 by Microsoft for use with the newly announced IBM personal computer. This first version consisted of about 4000 lines of assembler language code and ran in 8KB of memory. The original PC, which was based on the Intel 8088 chip, could address a maximum of 1 megabyte (1MB) of memory.

As the PC evolved, so did MS-DOS. Version 2, released in 1983, ran on the IBM PC/XT. It supported a hard disk drive, and incorporated many features that were similar to ones found in UNIX. Version 3, released in 1984, was designed for use with the new IBM PC/AT; this computer was based on the Intel 80286 chip. Version 4 (released in 1988), Version 5 (1991), and Version 6 (1993) provided further enhancements. They also included support for the more advanced CPU chips that were available (80386, 80486, and Pentium).

By modern standards, MS-DOS is technically obsolete. For example, it can run only one process at a time and can make only very limited use of memory

in excess of 1MB. However, it remains the most widely used operating system in the world, and there are a vast number of applications that run under its control. This software base helps to explain the continued popularity of MS-DOS, and the reluctance of many users to change to more modern systems.

Figure 6.32 shows the overall structure of MS-DOS. The BIOS (Basic Input/Output System) contains a collection of device drivers that correspond to the specific hardware components being used. MS-DOS performs input and output by invoking the BIOS routines; this isolates the rest of the operating system from the details of the hardware. The *kernel* provides service routines to implement memory management, process management, and the file system. The *shell* is the interface that interprets user commands and calls other operating system routines to execute them. MS-DOS provides a shell that interprets command lines and an alternative screen-oriented interface. Users can also install their own special-purpose shells.

MS-DOS does not support multiprogramming. A process can create a child process via a system call. However, the parent process is automatically suspended until the child process terminates. There may be many processes in memory at a particular time; however, only one of them can be active. All the rest will be blocked, waiting for a child to finish. Compare this with a multiprogramming system, in which the CPU can be switched among a number of active processes.

The lack of multiprogramming makes process management in MS-DOS relatively simple. Because of this restriction, however, MS-DOS cannot effectively support features such as background print spooling.

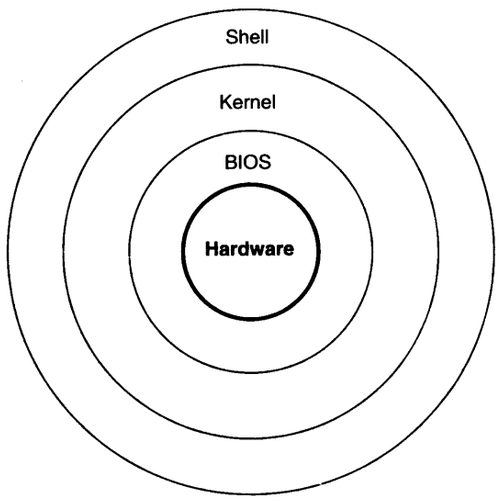


Figure 6.32 Structure of MS-DOS.

Figure 6.33 illustrates the memory model used in MS-DOS. There are four different areas of memory, each of which can be used in different ways. *Conventional memory* (the first 640KB) is where MS-DOS and application programs normally run. *Upper memory* usually contains device drivers (part of the BIOS). Some of the locations in this area of memory may actually be ROM (read-only memory) that is part of a BIOS chip or other special hardware.

The division between conventional memory and upper memory was part of the design of Version 1 of MS-DOS. Beginning with the 80286, the chips used in PCs were capable of addressing more than 1MB of memory. However, this limitation was retained in MS-DOS to provide compatibility with earlier hardware and software.

Because of the way MS-DOS is implemented, it can actually directly address up to 64KB beyond the original 1MB limit. (This is accomplished by having a segment register that contains an address close to 1MB, and specifying an offset of up to 65,535.) This additional 64KB of directly addressable memory is called *high memory*. The memory above 1MB (including the high memory area) is called *extended memory*.

MS-DOS cannot effectively use most of the extended memory to run programs. It is possible to load parts of MS-DOS into high memory; this makes more of the conventional memory addresses available for application programs. Parts of the extended memory can be mapped into unused addresses in the upper memory area, using the virtual-memory hardware on 80386 and higher CPUs. This makes locations in upper memory that are not needed for device drivers available for ordinary programs. MS-DOS can also treat extended memory as a separate device, in effect using it to simulate a disk.

These techniques are of some help in making more memory available to application programs. Several other methods have also been devised. However, the basic 1MB restriction still remains; this is one of the most severe limitations of MS-DOS.

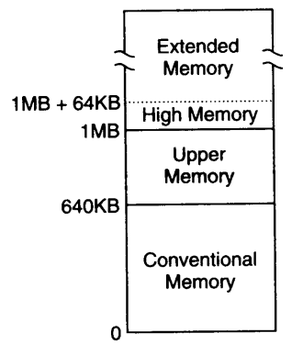


Figure 6.33 MS-DOS memory model.

Because it is intended for a single-user personal computer, MS-DOS does not provide the protection found on multi-user systems. Users are free to install their own interrupt-processing routines and device drivers. It is also possible for a program to manipulate system data structures to "trick" MS-DOS into providing undocumented services. These practices can add flexibility to the system, but also have the potential to cause serious problems.

Further information about MS-DOS can be found in Norton (1994) and Schulman (1993).

6.5.2 Windows 95

Windows 95 is the most recent version of Microsoft's "window-based" operating systems. Previous members of this family include Windows 3.1 and Windows NT. These systems provide a graphical user interface and desktop environment similar to those found on Apple's Macintosh computers. They are more advanced in design than MS-DOS, and enable programs to make better use of modern CPU hardware. For example, Windows 95 supports multiprogramming and allows user programs access to a very large virtual-memory space.

Figure 6.34 shows the overall structure of Windows 95. Most application programs run under the control of the *system virtual machine,* which provides hundreds of *application program interface* (API) functions. Windows 95 application programs can use a full 32-bit address space; these are referred to as *Win32 applications.* Support is also provided for older Windows applications

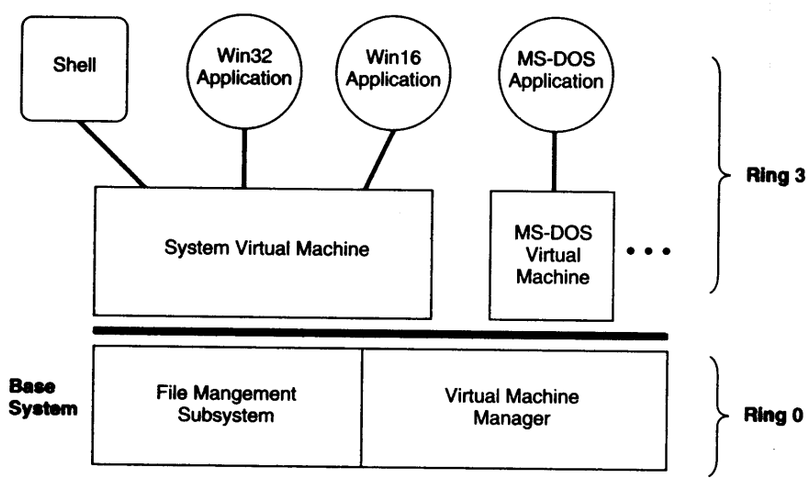


Figure 6.34 Overall structure of Windows 95.

that used a 16-bit address space (*Win16 applications*). The *shell* is a Win32 application that provides the essential user interface to the system.

Windows 95 also supports the execution of older MS-DOS programs. Each of these runs under control of a separate *MS-DOS virtual machine.* These virtual machines provide the same environment as a PC running MS-DOS, including the limitations on memory that can be addressed.

The Windows 95 virtual machines are supported by the *base system,* which is the actual underlying operating system. The x86 hardware provides four different levels of privilege, called *protection rings.* The base system runs in Ring 0 (the highest level of privilege), which is similar to the supervisor mode discussed earlier in this chapter. The virtual machines and application programs run in Ring 3, which generally corresponds to user mode. The intermediate levels of privilege (Rings 1 and 2) are not used in Windows 95.

The *virtual machine manager* is the most complex part of Windows 95, and the real "core" of the operating system. Each virtual machine (the system VM and MS-DOS VMs) has its own virtual address space and execution context (for example, register contents and current instruction address). There is also a set of resources available to applications running on each virtual machine. For example, Windows applications running on the system VM can call functions from the application programming interface. Applications running on an MS-DOS VM can use the MS-DOS interrupt interface, and may try to access the hardware directly. The virtual machine manager supports the virtual machines and provides fundamental low-level operating system services such as CPU scheduling, memory management, and interrupt handling.

Scheduling in Windows 95 is based on *processes* and *threads.* Each MS-DOS virtual machine is a process, and each Windows application running in the system virtual machine is a process. Thus a process in Windows 95 generally corresponds to an application program. Each process has its own memory space, execution context, and other resources.

A *thread* represents the execution of code associated with a process. Each process begins with one thread. MS-DOS and Win16 applications always have a single thread of execution. However, Win32 applications can create new threads to execute different parts of the code of the process concurrently. For example, a word processor could create a new thread to handle the printing of a document, while the user could continue editing using the original thread. Multiple threads within a process do not have separate memory spaces. Instead, each thread shares the memory and other resources of the parent process. This means that setting up a new thread, or switching between threads of the same process, is relatively efficient.

CPU scheduling is based on a system of priorities. At any given time, the highest-priority thread that is not blocked is scheduled for execution. If there is more than one highest-priority thread, they share the CPU in round-robin

fashion. The scheduler recalculates priorities for every thread in the system at fixed intervals. These priorities are based on several different considerations; the overall goal of the scheduling process is to provide a smoothly running system with good response time. Thus, this approach is similar to the intermediate-level scheduling we discussed in Section 6.3.2. For example, the priority of a thread may be temporarily raised because of a keystroke or mouse click to be processed by that thread. The increased priority gradually returns to its usual value over a short time interval.

There are several different levels of memory management in Windows 95. The system virtual machine runs in a virtual address space of 4 gigabytes (4GB). Each MS-DOS virtual machine runs in a virtual address space of 1MB. These virtual address spaces are all separate—each virtual machine runs in its own private address space, and is unable to interfere with any other VM.

Each Win32 application also runs in a 4GB address space. The addresses from 4MB to 2GB are private memory, which is not accessible by other applications. The rest of the address space is shared among all processes in the system VM. For example, the region from 2 to 3GB can be used to create memory mapped files that are shared between applications. The region from 3 to 4GB is reserved for the base system of Windows 95.

Win16 applications run in a similar 4GB address space. However, the same address space is shared by all Win16 applications—that is, there is no private memory for an application. This provides compatibility with previous versions of Windows, in which Win16 applications could refer to each others' memory. MS-DOS applications run in the 1MB virtual address space provided by the MS-DOS virtual machine.

All of these address spaces are kept separate by a system of virtual memory implemented with demand paging. When control is switched from one virtual machine or application to another, a different page map table is activated. Because of this, the same virtual address in different applications will be mapped into different physical locations in memory. Replacement of pages in the physical memory is governed by a least recently used (LRU) algorithm.

The Windows 95 file management system supports multiple file systems that can be concurrently accessed. It is a layered system, with an application program interface at the highest level. Additional components can be installed at many of the levels, to customize the file system and provide specialized services. Network support is also implemented via the file management system. This approach to file management is one of the major new components of Windows 95. In previous versions of Windows, file management services were provided by the underlying MS-DOS operating system.

Further information about Windows 95 can be found in Pietrek (1995) and King (1994).

6.5.3 SunOS

The SunOS operating system is the foundation of Sun's Solaris operating environment. Solaris consists of three major components: the SunOS operating system, the ONC distributed computing environment, and the OpenWindows development environment. ONC provides support for a distributed file system, a network naming service, and a remote procedure call facility. OpenWindows includes productivity tools and utilities that manage system resources, and provides an environment for developing and running application programs. SunOS is the underlying operating system that supports the other components of Solaris.

The original Solaris was developed for SPARC machines. Later versions, released in 1995 and 1996, can run on UltraSPARC, Pentium Pro, and PowerPC computers. This portability makes it easier to create applications for multiple hardware architectures and for networked systems that contain different types of machines. Further information about Solaris can be found in Becker et al. (1995).

This section focuses on the SunOS operating system, which is based on UNIX System V Release 4 (SVR4). We begin by discussing the development and design of UNIX, and then briefly survey the extensions included in SunOS.

The original versions of UNIX were developed at AT&T Bell Laboratories in the early 1970s. The first commercial release from Bell Labs was System III in 1982. However, several other versions were also in use, both within AT&T and by various universities and research labs. Microsoft's XENIX operating system was based on AT&T UNIX, with some features from other sources.

The University of California, Berkeley, became involved with UNIX in 1974. Several versions of the Berkeley System Distribution of UNIX (BSD UNIX) were developed and released. BSD UNIX was widely used in universities, and many computer vendors used it as a foundation for development of their own UNIX variants. For example, the original SunOS was based on version 4.2 BSD. Many of the enhancements developed at Berkeley were also incorporated into later releases of AT&T System V.

The existence of several different variants of UNIX created many problems for software developers. In the middle 1980s an IEEE working group, in conjunction with a group of UNIX users, proposed a set of standards for UNIX systems known as POSIX (Portable Operating System Interface for Computer Environments). The POSIX standards combined features from the most popular versions of UNIX into a single package. These standards have been endorsed by the National Institute of Standards and Technology as part of the Federal Information Processing Standard (FIPS).

UNIX System V Release 4 (SVR4) corresponds to the POSIX standards, and also to a number of other important standards. Thus, it represents a modern

unified version of the UNIX system. Unless otherwise stated, the following discussions in this section pertain to SVR4.

Figure 6.35 shows the overall structure of a UNIX system. The *kernel* isolates the rest of the system from the hardware, and provides services such as interrupt handling, memory management, file management, and process scheduling. Application programs interact with the kernel using a set of approximately 100 *system calls*. For example, a program may use a system call to open a file or to create a new child process.

The *shell* is the basic user interface to the system. When a user enters a command at a terminal, the shell interprets the command and calls the appropriate program or operating system routine. UNIX provides several hundred utility programs that perform operations such as text editing, sending mail, and displaying on-line documentation. These utilities are invoked via the shell; to the user, they appear to be commands built into the operating system.

Process scheduling in UNIX is governed by a system of priorities. By default, the scheduler implements a *time-sharing* policy. The goals of this policy are to give good response time to interactive processes and to provide good overall system throughput. With this time-sharing policy, all system processes have higher priority than user processes. Priorities of user processes depend on the amount of CPU time they have used. Processes that have used large amounts of CPU time are given lower priority than those that have used less time.

The scheduler also provides an optional *real-time* scheduling policy. With this policy, users can set fixed priorities for processes, which are not changed by the system. This allows real-time processes to respond quickly to time-

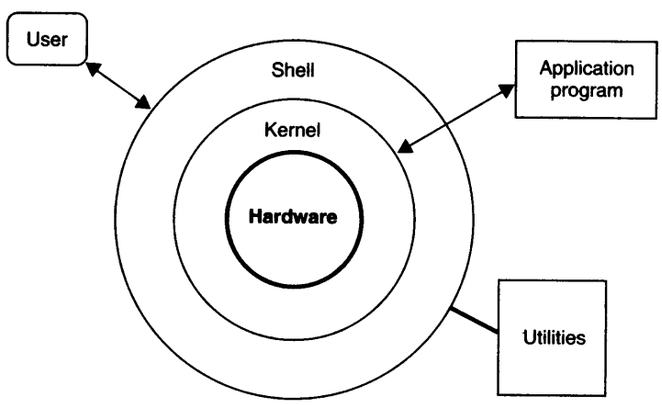


Figure 6.35 Overall structure of UNIX.

critical events. The highest-priority real-time process always gets the CPU as soon as it is ready to run, even if there are system processes waiting.

Processes can create other processes by using the *fork* system call. On return from this system call, the parent and child process have identical copies of their user-level context. The value returned by the *fork* call can be tested to determine which is the parent process and which is the child process. The *fork* is usually followed by an *exec* system call, which causes the child process to execute some other program or command. The parent process can synchronize its execution with the termination of the child process by using the *wait* system call.

UNIX also provides three other ways for processes to communicate with each other. One set of system calls allows processes to send and receive messages. A second set implements integer-valued semaphores that can be set and tested by different processes. A third group of system calls allows processes to define shared regions of memory.

Each process has a separate virtual address space. This virtual memory is implemented via demand paging. Pages in physical memory are selected for replacement using a modified working-set policy. A page that has not been referenced in a certain length of time is considered no longer to be in the working set of any process; thus it becomes a candidate for replacement. It is possible that the working sets of the processes being run might require more pages than can fit into memory. If this happens, one or more of the processes is temporarily suspended. Pages being used by these processes are removed from memory to make more room available.

UNIX organizes files into a hierarchical file system, with files grouped together under directories. Links can be established to allow a single file to be accessed by different names, or to appear in different directories. A *pipe* is a special type of file that can be used to connect the output of one process directly to the input of another. Physical devices are treated in the same way as files, and can be accessed using the same system calls.

The current version of SunOS adds several enhancements to the capabilities of SVR4. Symmetric multiprocessing is supported; multiple threads within the kernel can execute concurrently on different processors. Support is also provided for application-level multithreading. Extensions to the real-time scheduling policy have been implemented, with the goal of providing deterministic scheduling response. A number of security enhancements have been added, including several different user authentication modes. Tools are also provided to assist system administrators in monitoring and improving security.

Further information about SVR4 can be found in Rosen et al. (1990) and UNIX System Laboratories (1992). Additional specific information about SunOS can be found in Sun Microsystems (1995c).

6.5.4 UNICOS/mk

UNICOS/mk is Cray's operating system for the T3E multiprocessor. You may want to review the description of the T3E in Section 1.5.3 before proceeding.

The user and application program interfaces of UNICOS/mk are based on UNIX. UNICOS/mk complies with the latest POSIX standards, as well as with a number of other industry standards. However, the method of implementation is quite different from the normal UNIX structure discussed in the previous section (see Fig. 6.35).

Figure 6.36 shows the overall structure of UNICOS/mk. The system consists of a *microkernel* and a number of *servers*. The microkernel includes a minimal set of low-level services, and supports the basic machine-dependent aspects of the system. Thus it serves to isolate the rest of the system from the underlying hardware. The microkernel also provides a message-passing capability for transferring messages between the microkernel and the servers, and between the servers themselves.

The microkernel of UNICOS/mk is substantially smaller and simpler than the kernel of a conventional operating system. Most of the functions that are normally associated with the UNIX kernel are provided by specialized servers. Servers can run either in user space or in supervisor space. User space servers provide more insulation from possible errors in other servers, while supervisor space servers can offer better performance.

A T3E system is configured with two different types of processing elements—*user PEs* and *system PEs*. From a hardware perspective, these two types of PEs are identical. However, user PEs are dedicated to running application programs, and system PEs are dedicated to operating system services. The configuration also includes *redundant PEs*, which are set aside in case of the failure of a user PE.

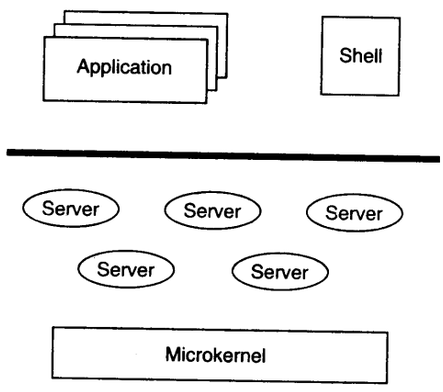


Figure 6.36 Overall structure of UNICOS/mk.

The functions of UNICOS/mk are distributed among user and system PEs. Each user PE contains a microkernel and a server that provides process management functions. Many other system requests, such as memory allocation, can also be handled locally. Global services, such as scheduling, file space allocation, security, and I/O management, are assigned to system PEs.

The T3E architecture was designed to be *scalable*. As we discussed in Section 1.5.3, processing elements can be added as needed to increase the computing power of the system. The distribution of operating system functions among the PEs also provides a scalable operating system environment. The number of system PEs increases as the size of the system increases. For every 16 user PEs there is, on average, one additional system PE. Thus the capability and capacity of the operating system grow to effectively support the overall system configuration.

The scalability of UNICOS/mk extends to support I/O operations as well. Global file servers on system PEs are used for functions such as opening and closing files. Local file servers on user PEs can perform direct servicing of read and write requests. However, a single "distributed" I/O request from one PE can also generate parallel data transfers that involve other PEs. The management of the overall file system can be distributed among multiple system PEs.

The T3E is a multiprocessor system that may have hundreds or thousands of PEs. However, UNICOS/mk presents users with a single "system image." For example, an interactive user logs into the system as though it were a single processor. During the interactive session, serial and parallel processes may run on many different PEs. Operating system functions may be performed using other system PEs. However, this distribution of work among PEs is completely transparent to the user.

Likewise, the T3E system is managed as a unified set of resources. UNICOS/mk is a single operating system distributed among the PEs, not a system replicated on each PE. This simplifies overall resource management, the users' view of files and other objects, and the system administrators' tasks.

Further information about UNICOS/mk can be found in Cray Research (1995c).

6.5.5 Amoeba*

The Amoeba operating system was developed at the Vrije Universiteit in Amsterdam. This research effort was aimed at understanding how to build a true distributed operating system. Amoeba connects multiple computers, of different types and at different locations, to provide the user with the illusion of a single powerful time-sharing system.

* Adapted from "Experiences with the Amoeba Distributed Operating System" by A. S. Tanenbaum et al., from *Communications of the ACM*, Vol. 33, No. 12, pp. 46–63, December 1990.

The Amoeba architecture consists of several different components, as illustrated in Fig. 6.37. The *workstations* are essentially intelligent terminals, on which users can carry out editing and other similar tasks. The *pool processors* are a group of CPUs that can be dynamically allocated to perform tasks, and then returned to the pool. For example, suppose that installing a certain complex program requires six different compilations. Six processors from the pool could be assigned to do these compilations in parallel. Similarly, an application such as a chess-playing program could use many pool processors at the same time to perform its computations.

Each of the specialized servers shown in Fig. 6.37 is dedicated to performing some specific function. For example, there are file servers, directory servers, and database servers. In some cases, there are multiple servers that can provide the same function. The gateways can be used to link Amoeba systems at different sites into a single uniform system. These gateways isolate Amoeba from the various protocols that must be used over wide area networks.

Amoeba is an object-based system. It can be viewed as a collection of *objects*, each of which supports a set of operations that can be performed. For a file object, for example, typical operations are reading, writing, appending, and deleting. Both hardware and software objects exist.

Access to objects is controlled by a system of *capabilities*. A capability is a kind of ticket that allows the holder of the capability to perform certain specified actions on an object. For example, a user process might have a capability for a file that permits it to read the file, but not to modify it. Capabilities are protected with special cryptographic checksums, to prevent users from tampering with them.

This object-based model is implemented using a *client-server* scheme. Associated with each object is a server process that manages that object. When

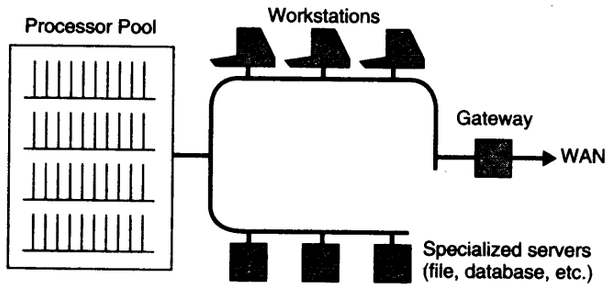


Figure 6.37 The Amoeba architecture (Adapted from "Experiences with the Amoeba Distributed Operating System" by A. S. Tanenbaum et al., from *Communications of the ACM*, Vol. 33, No. 12, pp. 46–63, December 1990. Copyright 1990, Association for Computing Machinery, Inc.).

a user process (the client) wants to perform an operation on an object, it sends a request message to the server that manages that object. The request contains a specification of the operation to be performed, and a capability that authorizes the client to perform that operation. The server sends a reply message when the requested operation has been completed. This combination of request and reply, together with the associated passing of parameters, is often referred to as *remote procedure call.*

Server processes are normally run on the specialized server machines shown in Fig. 6.37. However, a server is a logical part of the system, not necessarily related to any physical machine. In general, a client is unaware of the location at which a requested service is provided. The system administrator makes decisions about which servers are to run on which computers, and these decisions can be changed at any time. This allows for system tuning to improve performance, and for fault-tolerant operation in case of processor failure.

The Amoeba system itself is structured along the same lines as UNICOS/mk (see Fig. 6.36). All of the machines run the same *microkernel.* This microkernel handles the sending and receiving of messages, and provides some low-level memory management and process-scheduling functions. However, most of the operations that are normally associated with an operating system are provided by servers that manage objects.

For example, processes in Amoeba are implemented as objects. An existing process that wants to create a new child process begins by constructing a *process descriptor.* This descriptor contains information such as the type of hardware required and a description of the address space to be used by the new process. The descriptor is then sent to the memory and process server on the machine where the child process will be run. The server returns to the parent process a capability that allows it to control the execution of the child process.

Many of the servers in Amoeba are just ordinary user processes. For example, the file system server is a collection of user processes that manage file objects. Users who are not happy with the standard file system are free to write and use their own. This provides a high degree of flexibility.

Further information about Amoeba can be found in Tanenbaum (1992) and Tanenbaum et al. (1990).

EXERCISES

Section 6.2

1. In Section 6.2.1, we assumed that the occurrence of an interrupt inhibited all other interrupts of equal or lower priority. Would the scheme described in the text work if we simply inhibited all other interrupts of the same class?

2. Suppose the processing of a certain type of interrupt is unusually complex. It might not be desirable to leave other interrupts inhibited for the length of time required to complete the interrupt processing. Suggest a method of interrupt handling that would allow all interrupts to be enabled during most of the interrupt processing.

3. What are the advantages of having several different classes of interrupts, instead of just one class with flag bits to indicate the interrupt type?

4. Suppose there is a limit on the total amount of CPU time a job is allowed to use. This limit can vary from one job to another. What part of the operating system would be responsible for enforcing this time limit, and how might such a function be accomplished?

5. On SIC/XE, setting the IDLE bit of SW to 1 places the CPU into an idle status. Is such a hardware feature necessary on a computer that supports multiprogramming?

6. Suppose you wanted to implement a multiprogramming batch operating system on a computer that has no hardware interval timer. What problems might arise? Can you think of a way to solve these problems using some other hardware or software mechanism?

7. How would your answer to Exercise 6 change if the operating system also supported real-time processing?

8. Consider a multiprogramming operating system. Suppose there is only one user job currently ready to use the CPU. With the methods we have described, this user job would periodically be interrupted by the expiration of its time-slice. The status of the job would be saved; the dispatcher would then immediately restore this status and dispatch the job for another time interval. How might an operating system avoid this unnecessary overhead while still being able to service other jobs when they become ready?

9. Instead of maintaining a single list of all jobs with an indication of their status, some systems keep separate lists (i.e., a ready list, a blocked list, etc.). What are the advantages and disadvantages of this approach?

10. In the example shown in Fig. 6.15, we assumed that no timer interrupts occurred. Suppose the time-slice assigned to process P1 runs out between sequence numbers (2) and (3). Redraw the diagram, through the equivalent of sequence number (10), showing a possible series of events that might occur after the timer interrupt.

11. Redraw the diagram in Fig. 6.15, assuming that process P2 is given higher dispatching priority than process P1 and that preemptive process scheduling is used.

12. Suppose two jobs are being multiprogrammed together. Job A uses a great deal of CPU time and performs relatively little I/O. Job B performs many I/O operations, but requires very little CPU time. Which of these two jobs should be given higher dispatching priority to improve the overall system performance?

13. Suppose the I/O supervisor is able to select I/O requests from channel queues based on a priority system. Which of the two jobs in Exercise 12 should be given higher I/O priority in this way?

14. Suppose a certain I/O device can be reached via either one of two I/O channels; however, the device can be used by only one channel at a time. How would the I/O supervisor routines described in the text need to be changed to accommodate this situation?

15. How would you select the number and size of partitions for a system using fixed-partition memory management?

16. What are the advantages and disadvantages of the bounds-register approach to memory protection, as compared to the protection-key approach?

17. In a multiprogramming system, frequently an I/O operation is being performed for one job while another job is in control of the CPU. How could memory protection be provided for such an I/O operation? For example, how could one job be prevented from reading data into another job's partition?

18. Suppose a certain machine includes flag bits that indicate the type of each item stored in memory or in a register (for example, integer, character, floating-point number, instruction, or address pointer). How could this information be used to implement relocatable partitions on such a machine?

19. Is memory-protection hardware necessary on a machine that uses demand-paged memory management?

20. Is a relocating loader necessary on a machine that uses demand-paged memory management?

21. Some demand-paging systems select pages to be removed from memory based in part upon whether the page has been modified. That is, the system prefers to replace a page that has not been

modified since it was last loaded, instead of a page that has been modified. What are the advantages and disadvantages of such an approach?

22. What methods might a programmer use to improve the locality of reference of a program? What programming techniques and data structures would probably lead to poor locality of reference?

23. What would the diagram in Fig. 6.21(b) look like for a program with no locality of reference (i.e., a program in which each memory reference is independent of the previous references)? What would the diagram look like for a program with perfect locality of reference (i.e., one with each reference made to the same page as the previous reference)?

24. In a multiprogramming system, other jobs can perform useful work while one job is waiting for an I/O operation to complete. Why, then, does thrashing by one program in a virtual-memory system create a problem for other programs in the same system?

25. Outline algorithms for the four interrupt handlers on a SIC/XE multiprogramming operating system that uses demand paging.

26. Why was the timer interrupt assigned a lower priority than the SVC interrupt on the SIC/XE machine (see Section 6.2.1)?

27. Why was the I/O interrupt assigned a lower priority than the SVC interrupt on the SIC/XE machine?

Section 6.3

1. Give an algorithm for a file manager routine that performs the blocking and buffering operations illustrated in Fig. 6.23.

2. When might it be advantageous to use more than two buffers for a sequential file?

3. Draw a state-transition diagram similar to Fig. 6.7 for the three-level scheduling procedure illustrated in Fig. 6.24(b).

4. Is it possible for a job to have a shorter turnaround time under a multiprogramming operating system than under a single-job system for the same machine?

5. Describe algorithms and data structures for operating system routines that implement the request and release functions described in Section 6.3.3.

6. Suppose there is an event status block defined for each resource in the system in such a way that "the-event-has-occurred" is logically equivalent to "the-resource-is-free." Could the request and release functions described in Section 6.3.3 be implemented using the WAIT and SIGNAL operations described in Section 6.2.2?
7. Consider the two programs in Fig. 6.26. Instead of using request and release operations, we could simply inhibit timer interrupts between lines 24 and 27 of P1, and between lines 37 and 40 of P2. (The inhibiting and enabling of these interrupts could be done via operating system service calls.) Would this be a practical solution to the problem discussed in the text?
8. How might the operating system detect that a deadlock has occurred?

Section 6.4

1. What aspects of the underlying hardware must be simulated by a virtual machine manager? List as many items as you can, and briefly suggest methods for accomplishing the simulation.
2. Suppose you are designing an object-oriented operating system. What objects would be included in your system? List as many as you can, and also indicate what operations would be applicable to each type of object.
3. What would be the difference between a distributed operated system and a symmetric processing operating system running on a loosely coupled multiprocessor?
4. Suppose that additional processors are added to a master-slave multiprocessing system. What problems might arise because of this upgrade?
5. Suppose that a new machine is added to a network operating system. What effects would be noticed by the users of the system? What would the system administrator have to do to accomplish this upgrade?
6. Suppose that a new machine is added to a distributed operating system. What effects would be noticed by the users of the system? What would the system administrator have to do to accomplish this upgrade?

Chapter 7

Other System Software

In this chapter we present brief overviews of several different types of system software. The purpose of this chapter is to introduce the basic concepts and terms related to these pieces of software. Because of space limitations, we do not attempt to give detailed discussions of implementation. References are provided in each section for readers who want to study these topics further.

Section 7.1 describes the purpose and functions of a generalized database management system (DBMS), and discusses the relationship of the DBMS to the operating system. Section 7.2 gives a brief description of interactive text-editing systems, discussing the general approaches used in such editors. Section 7.3 introduces the topic of interactive program-debugging systems, and discusses some of the functions that such systems provide.

7.1 DATABASE MANAGEMENT SYSTEMS

This section describes the basic purpose and functions of a generalized database management system (DBMS). Our main focus in this section is the user's view of a DBMS. We also discuss how the DBMS functions are related to other types of software in the system. A comprehensive discussion of DBMS theory and implementation can be found in Date (1990).

Section 7.1.1 discusses the problems that led to the development of database management systems, and presents the basic concept of a DBMS. Section 7.1.2 describes in general terms how such systems appear to the user. Section 7.1.3 discusses the relationship of a DBMS to other system software, particularly the operating system.

7.1.1 Basic Concept of a DBMS

The development of database management systems resulted in large part from two major problems with conventional file-processing systems: *data redundancy* and *data dependence*. Consider for example, Fig. 7.1, which shows a simplified set of file-processing systems for a hypothetical university.

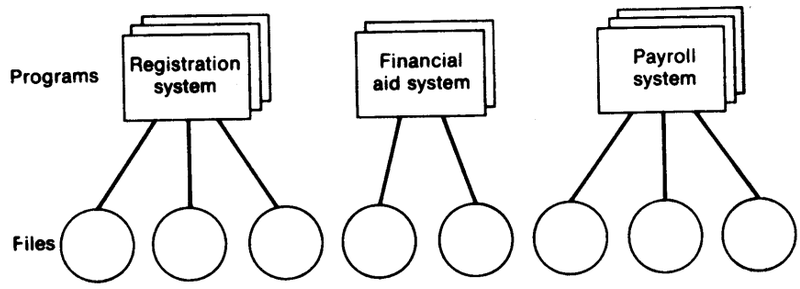


Figure 7.1 Separate file-processing systems.

This example includes a registration system to keep track of the enrollment of students in courses, a financial aid system to control payments of scholarship funds to students, and a payroll system for the entire university, including both faculty and student employees.

Each file-processing system consists of a set of programs and a set of data files. The format, organization, and content of these data files are specified by the person who initially defines the system. The files for each system contain only the information needed by that system; there is little or no coordination between files belonging to different systems.

The use of separate file-processing systems like those in Fig. 7.1 often led to a large amount of *data redundancy,* which is the duplication of data items in different files. For example, the number of courses in which a student is enrolled might be stored in one of the files for the registration system, and also in a file for the financial aid system. If a student were an employee of the university and a scholarship recipient, the student's name and address might appear in three different places.

The most obvious disadvantage of data redundancy is the additional storage space required. However, there are more serious problems associated with such duplication. A piece of information that is stored in several different places must be updated several times when its value changes. This multiple updating requires more computing time and more I/O operations. Even more serious is the possibility of inconsistent data because of program errors or differences in the schedules for updating the files. For example, an entry in a registration file might reflect the fact that a student had withdrawn from a certain course; however, this information might not be incorporated into the files for the financial aid system.

One possible solution to the problem of data redundancy is illustrated in Fig. 7.2(a). The information from all of the file systems is gathered into a single integrated *database* for the entire university. This database contains only one copy of each logical data item, so redundancy is eliminated. The database itself

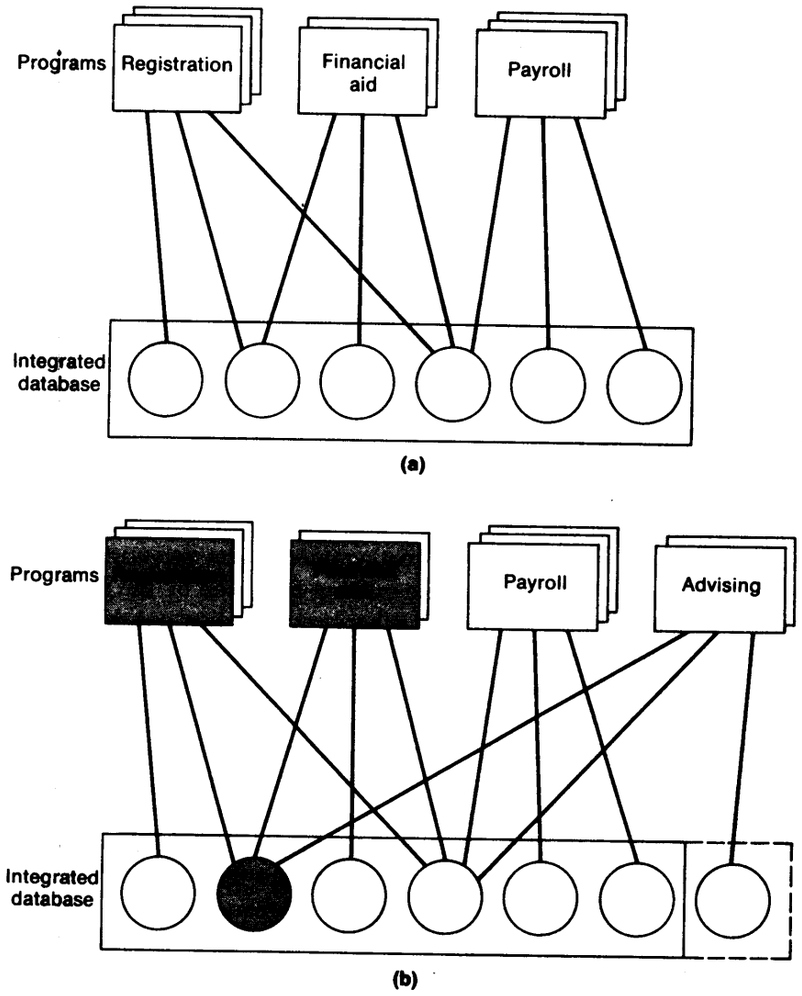


Figure 7.2 Data-dependent programs using an integrated database. (Shading indicates changes required by the addition of a new system.)

consists of a set of files. Different applications that require the same data item may share the file that contains the needed information.

Although the approach just described solves the problem of data redundancy, it may cause other difficulties. Application programs that deal directly with physical files are *data dependent,* which means they depend on characteristics such as record format and file organization and sequence.

Whenever these file characteristics are changed, the application programs must also be modified.

Suppose, for example, that a student-advising system is added to the set of applications. The new system may require certain information that is not already present in the database, so one or more new files may need to be created. On the other hand, some of the information required is already present. The advising system may need to refer to existing files for such items as each student's major code and current enrollment.

Unfortunately, this existing information may not be present in the form required by the new system. Suppose, for example, that the advising system must provide interactive access to enrollment information for all students who have a given advisor. This would require that the advisor's name be added to the enrollment information for a student and that the enrollment data be indexed by advisor. The content and organization of some existing files would need to be changed, and, because of data dependence, all other programs using these files would also have to be modified.

This situation is illustrated in Fig. 7.2(b). The advising system uses three database files: one new file and two existing ones. It was necessary to modify the format and structure of one of the existing files used by the new system (shown by shading in the figure). All other programs that use this file must therefore be changed. In this case, the changes involve programs in the existing registration and financial aid systems.

Problems like the one just described can be avoided by making application programs independent of details such as file organization and record format. Figure 7.3(a) shows how this can be accomplished. The user programs do not deal directly with the files of the database. Instead, they access the data by making requests to a *database management system* (DBMS). Application programs request data at a logical level, without regard for how the data is actually stored in files. For example, a program can request current enrollment information for a particular student. The DBMS determines which physical files are involved, and how these files are to be accessed, by referring to a stored *data mapping description.* It then reads the required records from the files of the database and converts the information into the form requested by the application program. This process is discussed in more detail in Sections 7.1.2 and 7.1.3.

The *data independence* provided by this approach means that file structures can be changed without affecting the application programs. Consider, for example, Fig. 7.3(b). As before, a new advising system has been added. A new file has been added to the database, and one existing file has been modified (indicated by the shading in the figure). The data mapping description has been modified to reflect these changes, but the application programs themselves remain unchanged. The same logical request from an application

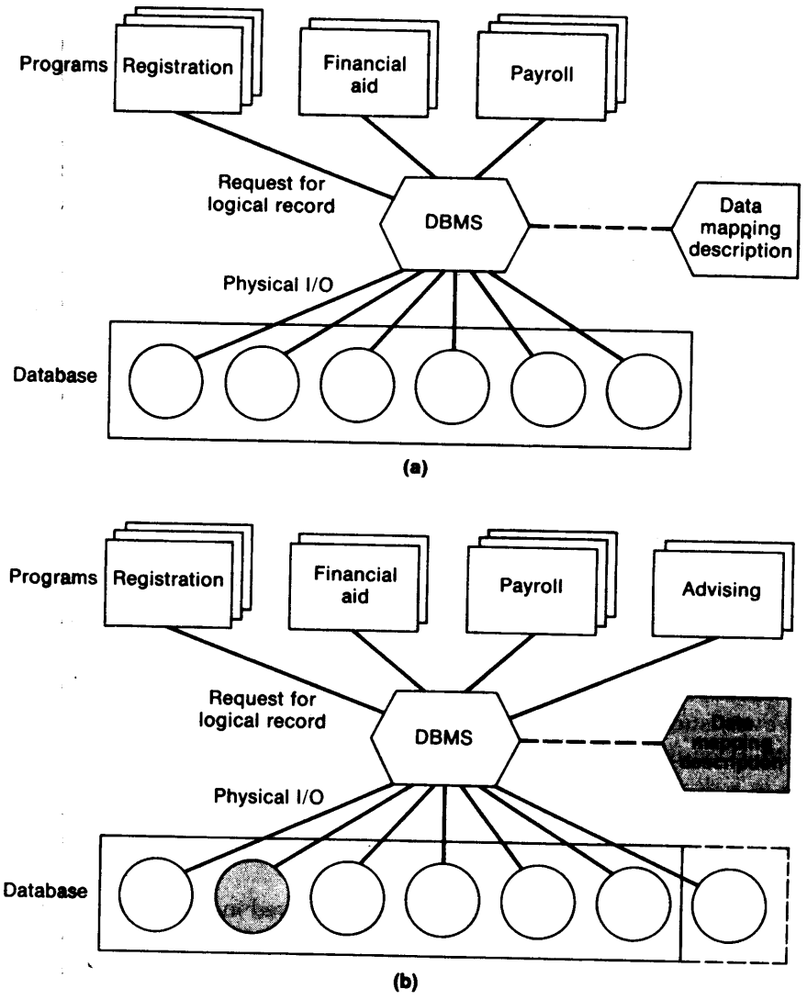


Figure 7.3 Data-dependent programs using a DBMS. (Shading indicates changes required by the addition of a new system.)

program may now result in a different set of operations on the files of the database. The application programs, however, are unaware of this difference because they are concerned only with logical items of information, not with how this information is stored in files.

The data independence provided by a DBMS is also important for other reasons. The techniques used for physical storage of the database can be

changed whenever it is desirable to do so. For example, some or all of the database can be moved to a different type of storage device. Files can be reorganized, sorted in different sequences, or indexed by a different set of keys. Decisions of this sort are usually made by a *database administrator*, who attempts to organize and store the data in a way that leads to the most efficient overall use of the database. All these changes can be made without affecting any of the application programs. Indeed, the programs are in general not able to detect that such changes have been made.

Because the data mapping description must be consulted for each reference to the database, using a DBMS involves more system overhead than running data-dependent programs. However, the benefits of data independence and reduced data redundancy usually outweigh the additional overhead required. This is particularly true when the content and structure of the database are subject to periodic changes because of new applications.

7.1.2 Levels of Data Description

As we discussed in the previous section, the use of a DBMS makes application programs independent of the way data is physically stored. With conventional file systems, the programmer is concerned with descriptions of the organization, sequencing, and indexing techniques of files to be processed. However, such details are not usually known to a programmer using a DBMS. Even if they are known, the programmer cannot rely on this knowledge in writing a program because the physical storage of the database may be changed at any time. Thus the application programmer's view of the data must be at a *logical* level, independent of file structure and other such questions of physical storage.

The information stored by a DBMS can be viewed in a number of different ways, depending upon the needs of the user. The most general such view is an overall logical database description called the *schema*. Figure 7.4 shows an example of such a schema. This example contains a number of logical database records, such as STUDENT and COURSE. Some of these records are connected by lines that indicate possible relationships between records of the given types. For example, the enrolled-in relationship specifies which students are enrolled in which courses. This example is intended to represent a database for a hypothetical university. The data items shown in each logical record illustrate the kinds of information that might be contained in such records. For a real database, there would probably be more logical record types, and each record would contain many more data items.

Database management systems differ considerably in the kinds of records and relationships that can be included in the schema. For example, some

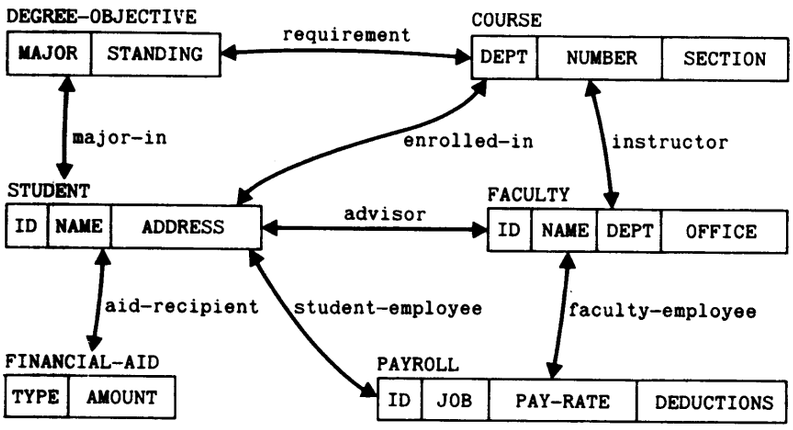


Figure 7.4 Schema for sample university database.

systems require that the logical database representation be expressed as a hierarchy or a tree structure, while others allow more general types of interconnections between records. Some systems do not allow explicit connections between records at all. These connections are expressed implicitly through the values of corresponding data items. A discussion of such *data models* is beyond the scope of this book. For further information and references, see Date (1990).

The schema gives a complete description of the logical structure and content of the database. However, most application programmers are concerned with only a small fraction of this information; a particular program usually deals with only a few types of records and relationships. The description of the data required by an application program is given by a *subschema.* There are usually many different subschemas corresponding to one schema. Each subschema gives a view of the database suited to the needs of the programs that use that subschema.

Figure 7.5 shows three different subschemas that correspond to various parts of the schema in Fig. 7.4. Subschema (a) is one that might be used by a program that produces a listing of students by major. To a program using subschema (a), the database appears to consist of a number of DEGREE-OBJECTIVE records, with each such record linked to a set of STUDENT records (one for each student with the given major). Subschema (b) might be used by a program that prints class rolls for each faculty member. This view of the database is a three-level structure. A record for each faculty member is linked to a set of records representing the courses he or she is teaching; each of these course records is similarly linked to a set of records representing the students enrolled in the course. Subschema (c) is designed for use by a program

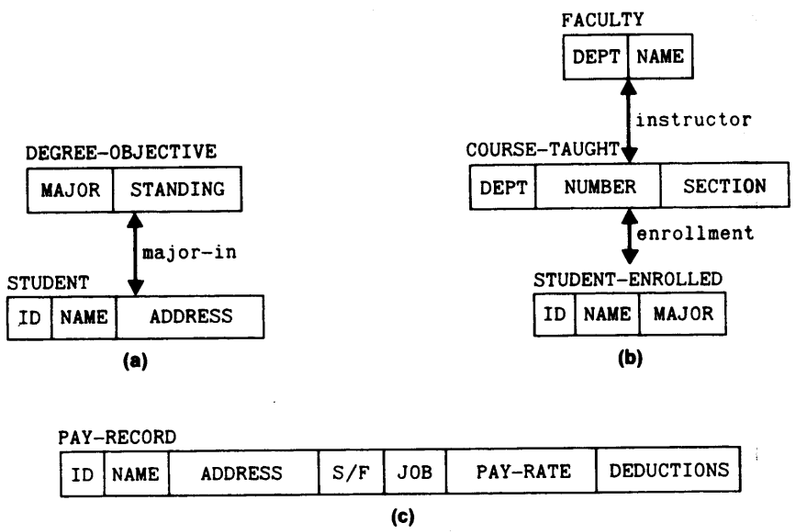


Figure 7.5 Three possible subschemas corresponding to the schema of Fig. 7.4.

that processes the payroll for the university. Each logical record in this subschema contains the information needed to issue a paycheck to an employee.

These three subschemas provide quite different views of the database. A subschema must be consistent with the schema—that is, it must be possible to derive the information in the subschema from the schema. Subschema (a) is simply a subset of the schema: the record names, data items, and relationships are the same as those contained in the corresponding part of the schema. This need not be true, however, for all subschemas.

In subschema (b), the application program uses record names that are different from those contained in the schema. It is also possible to use different names for individual data items. The FACULTY record in subschema (b) contains some, but not all, of the information from the FACULTY record of the schema. The COURSE-TAUGHT subschema record contains the same information that is present in the COURSE schema record. The STUDENT-ENROLLED record in the subschema contains information from the STUDENT record in the schema. However, the STUDENT-ENROLLED record also contains information about the student's major, which is contained in the DEGREE-OBJECTIVE record that is logically connected to the STUDENT record by the major-in relationship.

Subschema (c) consists of a single logical record type PAY-RECORD, whose data may come from three different schema records. Information concerning

rate of pay and deductions is taken from the PAYROLL schema record. Information such as employee name and address is obtained from either the STUDENT record or the FACULTY record, whichever is appropriate, by using the student-employee and faculty-employee relationships. The data field in PAY-RECORD that is designated S/F indicates whether the record pertains to a student employee or to a faculty member.

A subschema provides an application program with a view of the database that is suited to the needs of the particular program. The DBMS takes care of converting information from the database into the form specified by the subschema (see Section 7.1.3). As a result, the application program is simpler and easier to write because the programmer does not have to be concerned with data items and relationships that are not relevant to the application. The subschema is also an aid in providing data security because a program has no way of referring to data items not described in its subschema.

We have now discussed three different levels of data description in database management systems: the subschemas, the schema, and the data mapping description. A DBMS supplies languages, called *data description languages*, for defining the database at each of these levels. The subschemas are used by application programmers and are written in a *subschema description language* designed to be convenient for the programmer. Often subschema description languages are extensions of the data description capabilities in the programming language to be used. However, the subschemas are created and maintained by the database administrator. In defining a subschema, the database administrator must be sure that the view of data given in the subschema is derivable from the schema, and that it contains only those data items the application program is authorized to use.

The schema itself, and the physical data mapping description, are normally used only by the database administrator. On many systems, the *schema description language* is closely related to the subschema description language. It is also possible to use a more generalized language, because the schema is not used directly by application programmers. The *physical data description language* is influenced by the types of logical structures supported by the schema, and also by the types of files and storage devices supported by the DBMS.

Further discussions and examples of data description languages can be found in Date (1990).

7.1.3 Use of a DBMS

In the two preceding sections we introduced basic concepts and terminology related to database management systems. In this section we complete the picture by discussing how a user interacts with a DBMS, and how the DBMS is related to other pieces of system software.

The two principal methods for user interaction with a DBMS are illustrated in Fig. 7.6. The user can write a source program in the normal way, using a general-purpose programming language. However, instead of writing I/O statements of the form provided by the programming language, the programmer writes commands in a *data manipulation language* (DML) defined for use with the DBMS. These commands are often designed so that the DML appears to be merely an extension of the programming language. As shown in Fig. 7.6(a), a preprocessor may be used to convert the DML commands into programming language statements that call DBMS routines. The modified source program is then compiled in the usual way. Another approach is to modify the compiler to handle the DML statements directly. Some DMLs are defined as a set of CALL statements using the programming language itself, which avoids the need for preprocessing or compiler modification.

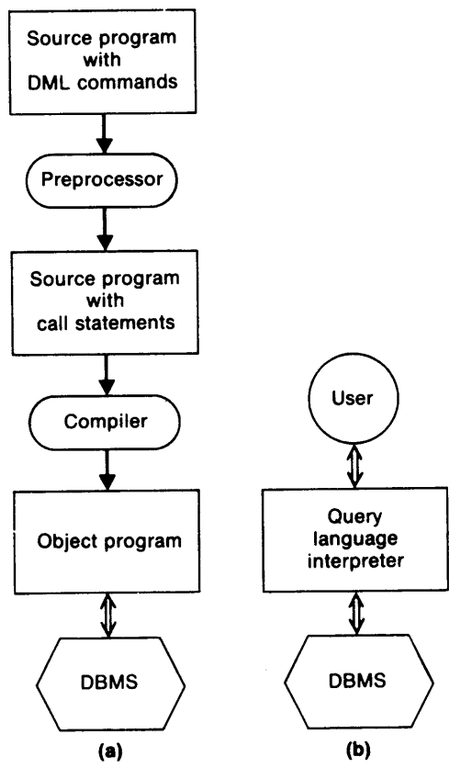


Figure 7.6 Interaction with a DBMS using (a) a data manipulation language and (b) a query language.

The other approach to DBMS interaction, illustrated in Fig. 7.6(b), does not require the user to write programs to access the database. Instead, users enter commands in a special *query language* defined by the DBMS. These commands are processed by a query-language interpreter, which calls DBMS routines to perform the requested operations.

Each of these approaches to user interaction has its own advantages. With a query language, it is possible to obtain results much more quickly because there is no need to write and debug programs. Query languages can also be used effectively by nonprogrammers, or by individuals who program only occasionally. Most query languages, however, have built-in limitations. For example, it may be difficult or impossible to perform a function for which the language was not designed. On the other hand, a DML allows the programmer to use all the flexibility and power of a general-purpose programming language; however, this approach requires much more effort from the user. Most modern database management systems provide both a query language and a DML so that a user can choose the form of interaction that best meets his or her needs. Further discussions and examples of DMLs and query languages can be found in Date (1990).

The sequence of operations performed by a DBMS in processing a request is essentially the same regardless of whether a DML or a query language is being used. These actions are illustrated in Fig. 7.7. The sequence of events begins when the DBMS is entered via a call from application program A (step 1 in the figure). If a query language is being used, program A is the query-language interpreter. We assume this call is a request to read data from the database. The sequences of events for other types of database operations are similar.

The request from program A is stated in terms of the subschema being used by A. For example, a program using the subschema in Fig. 7.5(c) might request the PAY-RECORD for a specified employee. To process such a request, the DBMS must first examine the subschema definition being used (step 2). The DBMS must also consider the relationship between the subschema and the schema (step 3) to interpret the request in terms of the overall logical database structure. Thus, for example, the DBMS would detect that it needed to read the schema PAYROLL record for the specified employee (see Fig. 7.4) to supply program A with its expected PAY-RECORD. In addition to this PAYROLL record, the DBMS would also need to examine the student-employee and faculty-employee relationships for the PAYROLL record in question, and read the corresponding STUDENT or FACULTY record.

After determining the logical database records that must be read (in terms of the schema), the DBMS examines the data mapping description (step 4). This operation gives the information needed to locate the required records in the files of the database. At this point, the DBMS has converted a logical request for a subschema record into physical requests to read data from one or more files. These requests for file I/O are passed to the operating system (step 5)

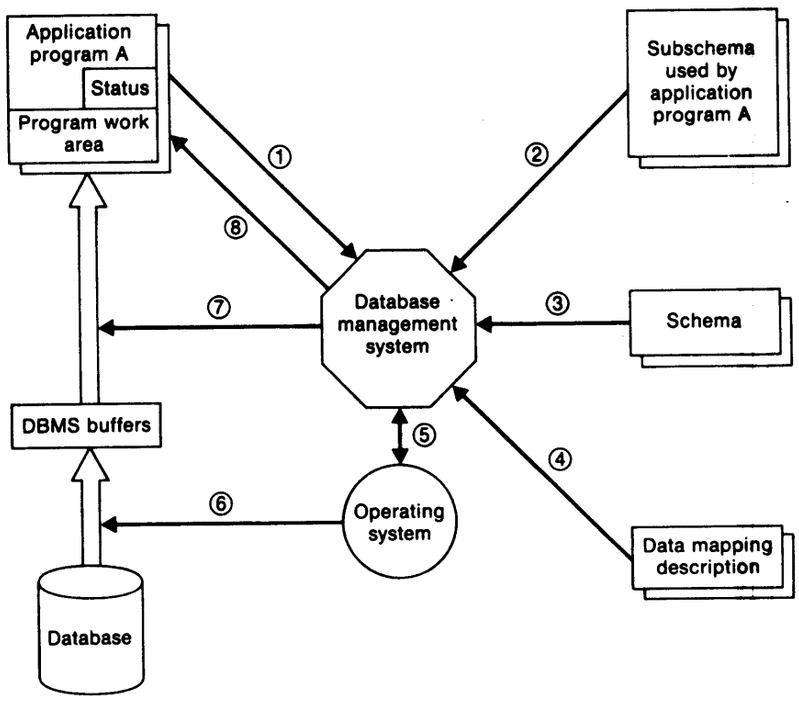


Figure 7.7 Typical sequence of actions performed by a DBMS. (Adapted from James Martin, *Computer Data-Base Organization*, 2nd ed., Copyright 1977, p. 83. Reprinted by permission of Prentice-Hall Inc., Englewood Cliffs, N.J.)

using the types of service calls discussed in Chapter 6. The operating system then issues channel and device commands to perform the necessary physical I/O operations (step 6). These I/O operations read the required records from the database into a DBMS buffer area.

After the physical I/O operations have been completed, all the data requested by the application program is present in central memory. However, this information must still be converted into the form expected by the program. The DBMS accomplishes this conversion (step 7) by again comparing the schema and the subschema. In the example we are discussing, the DBMS would extract data from the PAYROLL record and the associated STUDENT or FACULTY record, and construct the PAY-RECORD requested by program A. The PAY-RECORD would then be placed into a work area supplied by the application program; this completes the processing of the program's request for data. Finally, the DBMS returns control to the application program and makes available to the program a variety of status information, including any possible error indications.

Further details concerning the topics discussed in this section can be found in Date (1990).

7.2 TEXT EDITORS*

The interactive text editor has become an important part of almost any computing environment. No longer are editors thought of as tools only for programmers or for secretaries transcribing from marked-up copy generated by authors. It is now increasingly recognized that a text editor should be considered the primary interface to the computer for all types of "knowledge workers" as they compose, organize, study, and manipulate computer-based information.

In this section we briefly discuss interactive text-editing systems from the points of view of both the user and the system. Section 7.2.1 gives a general overview of the editing process. Section 7.2.2 expands upon this introduction by discussing various types of user interfaces and I/O devices. Section 7.2.3 describes the structure of a typical text editor and discusses a number of system-related issues.

7.2.1 Overview of the Editing Process

An *interactive editor* is a computer program that allows a user to create and revise a target document. The term *document* includes objects such as computer programs, text, equations, tables, diagrams, line art, and photographs—anything that one might find on a printed page. In this discussion, we restrict our attention to *text editors,* in which the primary elements being edited are character strings of the target text.

The document-editing process is an interactive user-computer dialogue designed to accomplish four tasks:

1. Select the part of the target document to be viewed and manipulated.
2. Determine how to format this view on-line and how to display it.
3. Specify and execute operations that modify the target document.
4. Update the view appropriately.

Selection of the part of the document to be viewed and edited involves first *traveling* through the document to locate the area of interest. This search is

*Adapted from Norman Meyrowitz and Andries van Dam, "Interactive Editing Systems: Part I and Part II," *ACM Computing Surveys,* September 1982. Copyright 1982, Association for Computing Machinery, Inc. These publications also contain much more detailed discussions of the editing process, descriptions of a large number of actual editors, and a comprehensive bibliography.

accomplished with operations such as *next screenful, bottom,* and *find pattern.* Traveling specifies where the area of interest is; the selection of what is to be viewed and manipulated there is controlled by *filtering.* Filtering extracts the relevant subset of the target document at the point of interest, such as the next screenful of text or the next statement. *Formatting* then determines how the result of the filtering will be seen as a visible representation (the *view*) on a display screen or other device.

In the actual *editing* phase, the target document is created or altered with a set of operations such as *insert, delete, replace, move,* and *copy.* The editing functions are often specialized to operate on *elements* meaningful to the type of editor. For example, a manuscript-oriented editor might operate on elements such as single characters, words, lines, sentences, and paragraphs; a program-oriented editor might operate on elements such as identifiers, keywords, and statements.

In a simple scenario, then, the user might travel to the end of the document. A screenful of text would be filtered, this segment would be formatted, and the view would be displayed on an output device. The user could then, for example, delete the first three words of this view.

7.2.2 User Interface

The user of an interactive editor is presented with a *conceptual model* of the editing system. This model is an abstract framework on which the editor and the world on which it operates are based. The conceptual model, in essence, provides an easily understood abstraction of the target document and its elements, with a set of guidelines describing the effects of operations on these elements. Some of the early *line editors* simulated the world of the keypunch. These editors allowed operations on numbered sequences of 80-character card-image lines, either within a single line or on an integral number of lines. Some more modern *screen editors* define a world in which a document is represented as a quarter-plane of text lines, unbounded both down and to the right. Operations manipulate portions of this quarter-plane without regard to line boundaries. The user sees, through a cutout, only a rectangular subset of this plane on a multiline display terminal. The cutout can be moved left or right, and up or down, to display other portions of the document.

Besides the conceptual model, the user interface is concerned with the *input devices,* the *output devices,* and the *interaction language* of the system. Brief discussions and examples of these aspects of the user interface are presented in the remainder of this section.

Input devices are used to enter elements of the text being edited, to enter commands, and to designate editable elements. These devices, as used with editors, can be divided into three categories: text devices, button devices, and

locator devices. *Text* or *string* devices are typically typewriter-like keyboards on which a user presses and releases keys, sending a unique code for each key. Virtually all current computer keyboards are of the QWERTY variety (named for the first six letters in the second row of the keyboard). Several alternative keyboard arrangements have been proposed, some of which offer significant advantages over the standard keyboard layout. None of these alternatives, however, seems likely to be widely accepted in the near future because of the retraining effort that would be required.

Button or *choice* devices generate an interrupt or set a system flag, usually causing invocation of an associated application-program action. Such devices typically include a set of special function keys on an alphanumeric keyboard or on the display itself. Alternatively, buttons can be simulated in software by displaying text strings or symbols on the screen. The user chooses a string or symbol instead of pressing a button.

Locator devices are two-dimensional analog-to-digital converters that position a cursor symbol on the screen by observing the user's movement of the device. The most common such devices for editing applications are the *mouse* and the *data tablet*. The data tablet is a flat, rectangular, electromagnetically sensitive panel. Either a ballpoint-pen-like *stylus* or a *puck*, a small device similar to a mouse, is moved over the surface. The tablet returns to a system program the coordinates of the position on the data tablet at which the stylus or puck is currently located. The program can then map these data-tablet coordinates to screen coordinates and move the cursor to the corresponding screen position. Locator devices usually incorporate one or more buttons that can be used to specify editing operations.

Text devices with arrow (cursor) keys can be used to simulate locator devices. Each of these keys shows an arrow that points up, down, left, or right, Pressing an arrow key typically generates an appropriate character sequence; the program interprets this sequence and moves the cursor in the direction of the arrow on the key pressed.

Voice-input devices, which translate spoken words to their textual equivalents, may prove to be the text input devices of the future. Voice recognizers are currently available for command input on some systems.

Formerly limited in range, output devices for editing are becoming more diverse. The output device lets the user view the elements being edited and the results of the editing operations. The first output devices were teletypewriters and other character-printing terminals that generated output on paper. Next, "glass teletypes" based on cathode ray tube (CRT) technology used the CRT screen essentially to simulate a hard-copy teletypewriter (although a few operations, such as backspacing, were performed more elegantly). Today's advanced CRT terminals use hardware assistance for such features as moving the cursor, inserting and deleting characters and lines, and scrolling lines and

pages. The more modern *professional workstations,* sometimes based on personal computers with high-resolution displays, support multiple proportionally spaced character fonts to produce realistic facsimiles of hard-copy documents. Thus the user can see the document portrayed essentially as it will look when printed on paper.

The interaction language of a text editor is generally one of several common types. The *typing-oriented* or *text command-oriented* method is the oldest of the major editor interfaces. The user communicates with the editor by typing text strings both for command names and for operands. These strings are sent to the editor and are usually echoed to the output device.

Typed specification often requires the user to remember the exact form of all commands, or at least their abbreviations. If the command language is complex, the user must continually refer to a manual or an on-line *help* function for a description of less frequently used commands. In addition, the typing required can be time consuming, especially for inexperienced users. The *function-key* interface addresses these deficiencies. Here each command has associated with it a marked key on the user's keyboard. For example, the *insert character* command might have associated with it a key marked IC. Function-key command specification is typically coupled with cursor-key movement for specifying operands, which eliminates much typing.

For the common commands in a function-key editor, usually only a single key is pressed. For less frequently invoked commands or options, an alternative textual syntax may be used. More commonly, however, special keys are used to shift the standard function-key interpretations, just as the SHIFT key on a typewriter shifts from lowercase to uppercase. As an alternative to shifting function keys, the standard alphanumeric keyboard is often *overloaded* to simulate function keys. For example, the user may press a *control* key simultaneously with a normal alphanumeric key to generate a new character that is interpreted like a function key.

Typing-oriented systems require familiarity with the system and language, as well as some expertise in typing. Function key-oriented systems often have either too few keys, requiring multiple-keystroke commands, or have too many unique keys, which results in an unwieldy keyboard. In either case, the function-key systems demand even more agility of the user than a standard keyboard does. The *menu-oriented* user interface is an attempt to address these problems. A *menu* is a multiple-choice set of text strings or *icons,* which are graphic symbols that represent objects or operations. The user can perform actions by selecting items from the menu. The editor prompts the user with a menu of only those actions that can be taken at the current state of the system.

One problem with a menu-oriented system can arise when there are many possible actions and several choices are required to complete an action. The display area for the menu is usually rather limited; therefore, the user might

be presented with several consecutive menus in a hierarchy before the appropriate command and its options appear. Since this can be annoying and detrimental to the performance of an experienced user, some menu-oriented systems allow the user to turn off menu control and return to a typing or function-key interface. Other systems have the most-used functions on a main command menu and have secondary menus to handle the less frequently used functions. Still other systems display the menu only when the user specifically asks for it. For example, the user might press a button on a mouse to display a menu with the full choice of applicable commands (perhaps temporarily overlaying some existing information on the screen). The mouse could be used to select the appropriate command. The system would then execute the command and delete the menu. Interfaces like this, in which prompting and menu information are given to the user at little added cost and little degradation in response time, are becoming increasingly popular.

7.2.3 Editor Structure

Most text editors have a structure similar to that shown in Fig. 7.8, regardless of the particular features they offer and the computers on which they are implemented. The *command language processor* accepts input from the user's input devices, and analyzes the tokens and syntactic structure of the commands. In this sense, the command language processor functions much like the lexical and syntactic phases of a compiler. Just as in a compiler, the command language processor may invoke semantic routines directly. In a text editor, these semantic routines perform functions such as editing and viewing.

Alternatively, the command language processor may produce an intermediate representation of the desired editing operations. This intermediate representation is then decoded by an interpreter that invokes the appropriate semantic routines. The use of an intermediate representation allows the editor to provide a variety of user-interaction languages with a single set of semantic routines that are driven from a common intermediate representation.

The semantic routines involve traveling, editing, viewing, and display functions. Editing operations are always specified explicitly by the user, and display operations are specified implicitly by the other three categories of operations. However, the traveling and viewing operations may be invoked either explicitly by the user or implicitly by the editing operations. The relationship between these classes of operations may be considerably more complicated than the simple model described in Section 7.2.1. In particular, there need not be a simple one-to-one relationship between what is currently displayed on the screen and what can be edited. To illustrate this, we take a closer look at some of the components of Fig. 7.8.

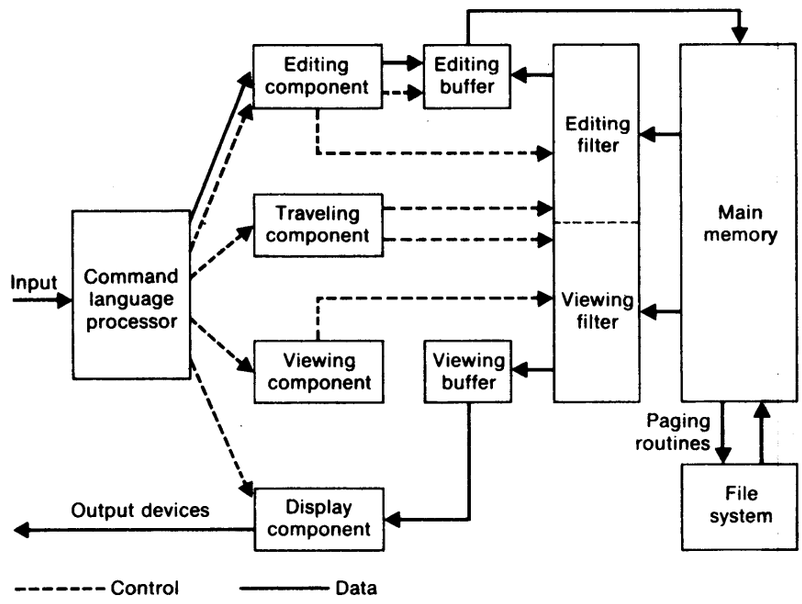


Figure 7.8 Typical editor structure. (Adapted from Norman Meyrowitz and Andries van Dam, "Interactive Editing Systems: Part I and Part II," *ACM Computing Surveys*, 1982. Copyright 1982, Association for Computing Machinery, Inc.)

In editing a document, the start of the area to be edited is determined by the *current editing pointer* maintained by the *editing component*, which is the collection of modules dealing with editing tasks. The current editing pointer can be set or reset explicitly by the user with traveling commands, such as *next paragraph* and *next screen*, or implicitly by the system as a side effect of the previous editing operation, such as *delete paragraph*. The *traveling component* of the editor actually performs the setting of the current editing and viewing pointers, and thus determines the point at which the viewing and/or editing filtering begins.

When the user issues an editing command, the editing component invokes the *editing filter*. This component filters the document to generate a new *editing buffer* based on the current editing pointer as well as on the editing filter parameters. These parameters, which are specified both by the user and the system, provide information such as the range of text that can be affected by an operation. Filtering may simply consist of the selection of contiguous characters beginning at the current point. Alternatively, filtering may depend on more complex user specifications pertaining to the content and structure of the document. Such filtering might result in the gathering of portions of the document

that are not necessarily contiguous. The semantic routines of the editing component then operate on the editing buffer, which is essentially a filtered subset of the document data structure. Note that this explanation is at a conceptual level—in a given editor, filtering and editing may be interleaved, with no explicit editing buffer being created.

Similarly, in viewing a document, the start of the area to be viewed is determined by the *current viewing pointer.* This pointer is maintained by the *viewing component* of the editor, which is a collection of modules responsible for determining the next view. The current viewing pointer can be set or reset explicitly by the user with a traveling command or implicitly by the system as a result of the previous editing operation. When the display needs to be updated, the viewing component invokes the *viewing filter.* This component filters the document to generate a new *viewing buffer* based on the current viewing pointer as well as on the viewing filter parameters. These parameters, which are specified both by the user and by the system, provide information such as the number of characters needed to fill the display, and how to select them from the document. In line editors, the viewing buffer may contain the current line; in screen editors, this buffer may contain a rectangular cutout of the quarter-plane of text. This viewing buffer is then passed to the *display component* of the editor, which produces a display by mapping the buffer to a rectangular subset of the screen, usually called a *window.*

The editing and viewing buffers, while independent, can be related in many ways. In the simplest case, they are identical: the user edits the material directly on the screen (see Fig. 7.9). On the other hand, the editing and viewing buffers may be completely disjoint. For example, the user of a certain editor

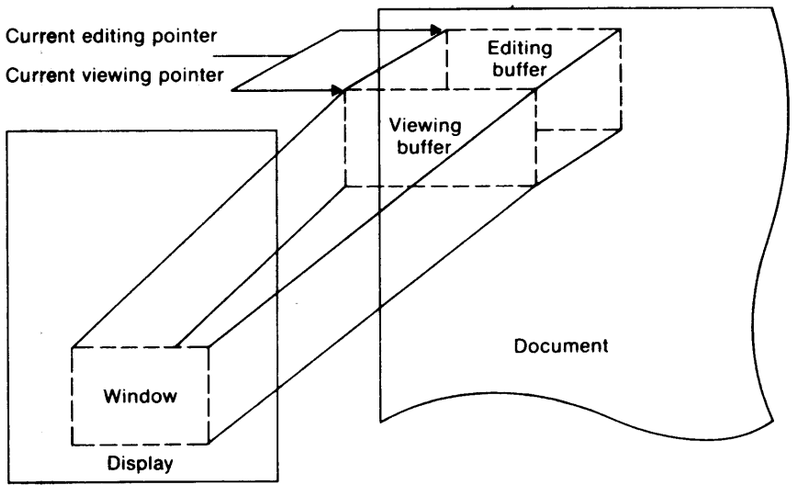


Figure 7.9 Simple relationship between editing and viewing buffers.

might travel to line 75, and after viewing it, decide to change all occurrences of "ugly duckling" to "swan" in lines 1 through 50 of the file by using a *change* command such as

```
[1,50] c/ugly duckling/swan/
```

As part of this editing command, there is implicit travel to the first line of the file. Lines 1 through 50 are then filtered from the document to become the editing buffer. Successive substitutions take place in this editing buffer without corresponding updates of the view. If the pattern is found, the current editing and viewing pointers are moved to the last line on which it is found, and that line becomes the default contents of both the editing and viewing buffers. If the pattern is not found, line 75 remains in the editing and viewing buffers.

The editing and viewing buffers can also partially overlap, or one may be completely contained in the other. For example, the user might specify a search to the end of the document, starting at a character position in the middle of the screen. In this case the editing filter creates an editing buffer that contains the document from the selected character to the end of the document. The viewing buffer contains the part of the document that is visible on the screen, only the last part of which is in the editing buffer.

Windows typically cover either the entire screen or a rectangular portion of it. Mapping viewing buffers to windows that cover only part of the screen is especially useful for editors on modern graphics-based workstations. Such systems can support multiple windows, simultaneously showing different portions of the same file or portions of different files. This approach allows the user to perform inter-file editing operations much more effectively than with a system having only a single window.

The mapping of the viewing buffer to a window is accomplished by two components of the system. First, the viewing component formulates an ideal view, often expressed in a device-independent intermediate representation. This view may be a very simple one consisting of a window's worth of text arranged so that lines are not broken in the middle of words. At the other extreme, the idealized view may be a facsimile of a page of fully formatted and typeset text with equations, tables, and figures. Second, the *display component* takes this idealized view from the viewing component and maps it to a physical output device in the most efficient manner possible.

Updating of a full-screen display connected over low-speed lines is slow if every modification requires a full rewrite of the display surface. Greatly improved performance can be obtained by using optimal screen-updating algorithms. These algorithms are based on comparing the current version of the screen with the following screen. They make use of the innate capabilities of the terminal, such as *insert-character* and *delete-character* functions, transmitting only those characters needed to generate a correct display.

Device-independent output, like device-independent input, helps provide portability of the interaction language. Decoupling editing and viewing operations from display functions for output avoids the need to have a different version of the editor for each output device. Many editors make use of a *terminal-control database.* Instead of having explicit terminal-control sequences in the display routines, these editors simply call terminal-independent library routines such as *scroll down* or *read cursor position.* These library routines use the terminal-control database to look up the appropriate control sequences for a particular terminal. Consequently, adding a new terminal merely entails adding a database description of that terminal.

The components of the editor deal with a user document on two levels: in main memory and in the disk file system. Loading an entire document into main memory may be infeasible. However, if only part of a document is loaded, and if many user-specified operations require a disk read by the editor to locate the affected portion, editing might be unacceptably slow. In some systems, this problem is solved by mapping the entire file into virtual memory and letting the operating system perform efficient demand paging. An alternative is to provide *editor paging routines,* which read one or more logical portions of a document into memory as needed. Such portions are often termed *pages,* although there is usually no relationship between these pages and hardcopy document pages or virtual-memory pages. These pages remain resident in main memory until a user operation requires that another portion of the document be loaded.

Documents are often represented internally not as sequential strings of characters, but in an *editor data structure* that allows addition, deletion, and modification with a minimum of I/O and character movement. When stored on disk, the document may be represented in terms of this data structure or in an editor-independent general-purpose format, which might consist of character strings with embedded control characters such as *linefeed* and *tab.*

Editors function in three basic types of computing environments: time sharing, stand-alone, and distributed. Each type of environment imposes some constraints on the design of an editor. The time-sharing editor must function swiftly within the context of the load on the computer's processor, central memory, and I/O devices. The editor on a stand-alone system must have access to the functions that the time-sharing editor obtains from its host operating system. These may be provided in part by a small local operating system, or they may be built into the editor itself if the stand-alone system is dedicated to editing. The editor operating in a distributed resource-sharing local network must, like a stand-alone editor, run independently on each user's machine and must, like a time-sharing editor, contend for shared resources such as files.

Some time-sharing editing systems take advantage of hardware capabilities to perform editing tasks. Many workstations and intelligent terminals have their

own microprocessors and local buffer memories in which editing manipulations can be done. Small actions are not controlled by the CPU of the host processor, but are handled by the local workstation. For example, the editor might send a full screen of material from the host processor to the workstation. The user would then be free to add and delete characters and lines, using the local buffer and workstation-based control commands. After the buffer has been edited in this way, its updated contents would be transmitted back to the host processor.

The advantage of this scheme is that the host need not be concerned with each minor change or keystroke; however, this is also the major disadvantage. With a nonintelligent terminal, the CPU sees every character as it is typed and can react immediately to perform error checking, to prompt, to update the data structure, etc. With an intelligent workstation, the lack of constant CPU intervention often means that the functionality provided to the user is more limited. Also, local editing operations on the workstation may be lost in the event of a system crash. On the other hand, systems that allow each character to interrupt the CPU may not use the full hardware editing capabilities of the workstation because the CPU needs to see every keystroke and provide character-by-character feedback.

7.3 INTERACTIVE DEBUGGING SYSTEMS*

An interactive debugging system provides programmers with facilities that aid in the testing and debugging of programs. A number of such systems are now available; however, the extent to which these tools are used varies widely. This section describes some of the most important requirements for an interactive debugging system, and discusses some basic system considerations involved in providing these services. The discussion is deliberately broad in scope, and is not limited to any particular existing system.

Section 7.3.1 presents a brief introduction to the most important functions and capabilities of an interactive debugging system and discusses some of the problems involved. Section 7.3.2 describes how the debugging tool should be related to other parts of the system. Section 7.3.3 discusses the nature of the user interface for an interactive debugger.

7.3.1 Debugging Functions and Capabilities

This section describes some of the most important capabilities of an interactive debugging system. Some of these functions are much more difficult to implement than others, and in a few cases, the best form of solution is not yet clear.

*Adapted from Rich Seidner and Nick Tindall, "Interactive Debug Requirements." *Software Engineering Notes* and *SIGPLAN Notices*, August 1983. Copyright 1983, Association for Computing Machinery, Inc.

The most obvious requirement is for a set of *unit test* functions that can be specified by the programmer. One important group of such functions deals with *execution sequencing*, which is the observation and control of the flow of program execution. For example, the program may be halted after a fixed number of instructions are executed. Similarly, the programmer may define *breakpoints* which cause execution to be suspended when a specified point in the program is reached. After execution is suspended, other debugging commands can be used to analyze the progress of the program and to diagnose errors detected. Execution of the program can then be resumed. As an alternative, the programmer can define conditional expressions that are continually evaluated during the debugging session. Program execution is suspended when any of these conditions becomes true. Given a good graphical representation of program progress, it may even be useful to run the program at various speeds, called *gaits*.

A debugging system should also provide functions such as tracing and traceback. *Tracing* can be used to track the flow of execution logic and data modifications. The control flow can be traced at different levels of detail: procedure, branch, individual instruction, and so on. The tracing can also be based on conditional expressions as previously mentioned. *Traceback* can show the path by which the current statement was reached. It can also show which statements have modified a given variable or parameter. This kind of information should be displayed symbolically, and should be related to the source program—for example, as statement numbers rather than as hexadecimal displacements.

It is also important for a debugging system to have good *program-display* capabilities. It must be possible to display the program being debugged, complete with statement numbers. The user should be able to control the level at which this display occurs. For example, the program may be displayed as it was originally written, after macro expansion, and so on. It is also useful to be able to modify and incrementally recompile the program during the debugging session. The system should save all the debugging specifications (breakpoint definitions, display modes, etc.) across such a recompilation, so the programmer does not need to reissue all of these debugging commands. It should be possible to symbolically display or modify the contents of any of the variables and constants in the program, and then resume execution. The intent is to give the appearance of an interpreter, regardless of the underlying mechanism that is actually used.

Many other functions and capabilities are commonly found in interactive debugging systems. Further descriptions of such features can be found in Lazzerini (1992).

In providing functions such as those just described, a debugging system should consider the language in which the program being debugged is written. Most user environments, and many applications systems, involve the use of several different programming languages. What is needed is a single debugging tool that is applicable to such multilingual situations. Debugger commands that

initiate actions and collect data about a program's execution should be common across languages. However, a debugging system must be sensitive to the specific language being debugged so that procedural, arithmetic, and conditional logic can be coded in the syntax of that language.

These requirements have a number of consequences for the debugger and for other software. When the debugger receives control, the execution of the program being debugged is temporarily suspended. The debugger must then be able to determine the language in which the program is written and set its *context* accordingly. Likewise, the debugger should be able to switch its context when a program written in one language calls a program written in a different language. To avoid confusion, the debugger should inform the user of such changes in context.

The context being used has many different effects on the debugging interaction. For example, assignment statements that change the values of variables during debugging should be processed according to the syntax and semantics of the source programming language. A COBOL user might enter the debugging command MOVE 3.5 TO A, whereas a FORTRAN user might enter A = 3.5. Likewise, conditional expressions should use the notation of the source language. The condition that A be unequal to B might be expressed as IF A NOT EQUAL TO B for debugging a COBOL program, and as IF (A .NE. B) for a FORTRAN program. Similar differences exist with respect to the form of statement labels, keywords, and so on.

The notation used to specify certain debugging functions varies according to the language of the program being debugged. However, the functions themselves are accomplished in essentially the same manner regardless of the source programming language. To perform these operations, the debugger must have access to information gathered by the language translator. However, the internal symbol dictionary formats often vary widely between different language translators; the same is true for statement-location information. Future compilers and assemblers may aim toward a consistent interface with the debugging system. One approach is for the language translators to produce the needed information in a standard external form for the debugger regardless of the internal form used in the translator. Another possibility is for the language translator to provide debugger interface modules that can respond to requests for information in a standard way regardless of the language being debugged.

A similar issue is related to the display of source code during the debugging session. Again, there are two main options. The language translator may provide the source code or source listing tagged in some standard way so that the debugger has a uniform method of navigating about it. Alternatively, the translator may supply an interface module that does the navigation and display in response to requests from the debugger.

It is also important that a debugging system be able to deal with optimized code. Application code used in production environments is usually optimized. It is not enough to return to unoptimized forms of the code, because the problem will often disappear. However, the requirement for handling optimized code may create many problems for the debugger. We briefly describe some of these difficulties. Further discussions can be found in Lazzerini (1992).

Many optimizations involve the rearrangement of segments of code in the program. For example, invariant expressions can be removed from loops. Separate loops can be combined into a single loop, or a loop may be partially unrolled into straight-line code. Redundant expressions may be eliminated; in some cases, this may cause entire statements to disappear. Blocks of code may be rearranged to eliminate unnecessary branch instructions, which provides more efficient execution. See Section 5.3.2 for examples of some of these transformations.

All these types of optimization create problems for the debugger. The user of a debugging system deals with the source program in its original form, before optimizations are performed. However, code rearrangement alters the execution sequence and may affect tracing, breakpoints, and even statement counts if entire statements are involved. If an error occurs, it may be difficult to relate the error to the appropriate location in the original source program.

A different type of problem occurs with respect to the storage of variables. When a program is translated, the compiler normally assigns a *home location* in main memory (or in an activation record) to each variable. However, as we discussed in Section 5.2.2, variable values may be temporarily held in registers at various times to improve speed of access. Statements referring to these variables use the value stored in the register, instead of taking the variable value from its home location. These optimizations present no problem for displaying the values of such variables. However, if the user changes the value of the variable in its home location while debugging, the modified value might not be used by the program as intended when execution is resumed. In a similar type of global optimization, a variable may be permanently assigned to a register. In this case, there may be no home location at all.

The debugging of optimized code requires a substantial amount of cooperation from the optimizing compiler. In particular, the compiler must retain information about any transformations that it performs on the program. Such information can be made available both to the debugger and to the programmer. Where reasonable, the debugger should use this information to modify the debugging request made by the user, and thereby perform the intended operation. For example, it may be possible to simulate the effect of a breakpoint that was set on an eliminated statement. Similarly, a modified variable value can be stored in the appropriate register as well as at the home location for that variable. However, some more complex optimizations cannot be handled as easily.

In such cases, the debugger should merely inform the user that a particular function is unavailable at this level of optimization, instead of attempting some incomplete imitation of the function.

7.3.2 Relationship with Other Parts of the System

An interactive debugger must be related to other parts of the system in many different ways. One very important requirement for an interactive debugger is that it always be available. This means that it must appear to be a part of the run-time environment and an integral part of the system. When an error is discovered, immediate debugging must be possible because it may be difficult or impossible to reproduce the program failure in some other environment or at some other time. Thus the debugger must communicate and cooperate with other operating system components such as interactive subsystems.

For example, users need to be able to debug in a production environment. Debugging is even more important at production time than it is at application-development time. When an application fails during a production run, work dependent on that application stops. Since the production environment is often quite different from the test environment, many program failures cannot be repeated outside the production environment.

The debugger must also exist in a way that is consistent with the security and integrity components of the system. It must not be possible for someone to use the debugger to access any data or code that would not otherwise be accessible to that individual. Similarly, it must not be possible to use the debugger to interfere with any aspect of system integrity. Use of the debugger must be subject to the normal authorization mechanisms and must leave the usual audit trails. One benefit of the debugger, at least by comparison with a storage dump, is that it controls the information that is presented. Whereas a dump may include information that happens to have been left in storage, the debugger presents information only for the contents of specific named objects.

The debugger must coordinate its activities with those of existing and future language compilers and interpreters, as described in the preceding section. It is assumed that debugging facilities in existing languages will continue to exist and be maintained. The requirements for a cross-language debugger assume that such a facility would be installed as an alternative to the individual language debuggers.

7.3.3 User-Interface Criteria

The behavior and presentation of an interactive system is crucial to its acceptance by users. Probably the most common complaint about debugging products is