```
repeat
      ...
     produce an item in nextp
      ...
    wait(empty);
    wait(mutex);
      ...
    add nextp to buffer
      ...
    signal(mutex);
    signal(full);
until false;
```

Figure 6.10 The structure of the producer process.

only reading as *readers,* and to the rest as *writers.* Obviously, if two readers access the shared data object simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the shared object simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to as the *readers–writers* problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers–writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply

```
repeat
    wait(full);
    wait(mutex);
      ...
    remove an item from buffer to nextc
      ...
    signal(mutex);
    signal(empty);
      ...
    consume the item in nextc
      ...
until false;
```

Figure 6.11 The structure of the consumer process.

```
wait(wrt);
    ...
  writing is performed
    ...
signal(wrt);
```

Figure 6.12 The structure of a writer process.

because a writer is waiting. The *second* readers–writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

We note that a solution to either problem may result in *starvation*. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we present a solution to the first readers–writers problem. Refer to the Bibliographic Notes for relevant references on starvation-free solutions to the readers–writers problem.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
var mutex, wrt: semaphore;
    readcount : integer;
```

The semaphores *mutex* and *wrt* are initialized to 1; *readcount* is initialized to 0. The semaphore *wrt* is common to both the reader and writer processes. The *mutex* semaphore is used to ensure mutual exclusion when the variable *readcount* is updated. *Readcount* keeps track of how many processes are currently reading the object. The semaphore *wrt* functions as a mutual exclusion semaphore for the writers. It also is used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 6.12; the code for a reader process is shown in Figure 6.13. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on *wrt*, and $n - 1$ readers are queued on *mutex*. Also observe that, when a writer executes *signal*(*wrt*), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

6.5.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a

```
wait(mutex);
   readcount := readcount + 1;
   if readcount = 1 then wait(wrt);
signal(mutex);
      ...
    reading is performed
      ...
wait(mutex);
   readcount := readcount - 1;
   if readcount = 0 then signal(wrt);
signal(mutex);
```

Figure 6.13 The structure of a reader process.

bowl of rice, and the table is laid with five single chopsticks (Figure 6.14). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The dining-philosophers problem is considered a classic synchronization problem, neither because of its practical importance, nor because computer scientists dislike philosophers, but because it is an example for a large class of concurrency-control problems. It is a simple

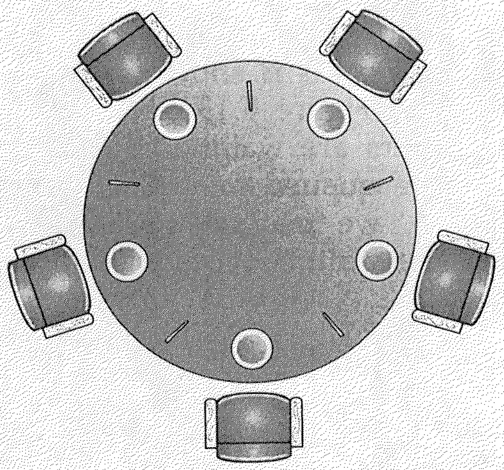


Figure 6.14 The situation of the dining philosophers.

representation of the need to allocate several resources among several processes in a deadlock and starvation-free manner.

One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a *wait* operation on that semaphore; she releases her chopsticks by executing the *signal* operation on the appropriate semaphores. Thus, the shared data are

```
var chopstick: array [0..4] of semaphore;
```

where all the elements of *chopstick* are initialized to 1. The structure of philosopher *i* is shown in Figure 6.15.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating a deadlock. Suppose that all five philosophers become hungry simultaneously, and each grabs her left chopstick. All the elements of *chopstick* will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next. In Section 6.7, we present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (note that she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

```
repeat
    wait(chopstick[i]);
    wait(chopstick[i+1 mod 5]);
        ...
        eat
        ...
    signal(chopstick[i]);
    signal(chopstick[i+1 mod 5]);
        ...
        think
        ...
until false;
```

Figure 6.15 The structure of philosopher *i*.

Finally, any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

6.6 ■ Critical Regions

Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect use can still result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place, and these sequences do not always occur.

We have seen an example of such types of errors in the use of counters in our solution to the producer–consumer problem (Section 6.1). In that example, the timing problem happened only rarely, and even then the counter value appeared to be a reasonable value — off by only 1. Nevertheless, this solution is obviously not an acceptable one. It is for this reason that semaphores were introduced in the first place.

Unfortunately, such timing errors can still occur with the use of semaphores. To illustrate how, let us review the solution to the critical-section problem using semaphores. All processes share a semaphore variable *mutex*, which is initialized to 1. Each process must execute *wait*(*mutex*) before entering the critical section, and *signal*(*mutex*) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.

Let us examine the various difficulties that may result. Note that these difficulties will arise even if a *single* process is not well behaved. This situation may be the result of an honest programming error or of an uncooperative programmer.

- Suppose that a process interchanges the order in which the *wait* and *signal* operations on the semaphore *mutex* are executed, resulting in the following execution:

```
signal(mutex);
    ...
  critical section
    ...
wait(mutex);
```

  In this situation, several processes may be executing in their critical section simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a process replaces *signal*(*mutex*) with *wait*(*mutex*). That is, it executes

  ```
  wait(mutex);
     ...
     critical section
     ...
  wait(mutex);
  ```

  In this case, a deadlock will occur.

- Suppose that a process omits the *wait*(*mutex*), or the *signal*(*mutex*), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when semaphores are used incorrectly to solve the critical-section problem. Similar problems may arise in the other synchronization models we discussed in Section 6.5.

To deal with the type of errors we have outlined, a number of high-level language constructs have been introduced. In this section, we describe one fundamental high-level synchronization construct — the *critical region* (sometimes referred to as *conditional critical region*). In Section 6.7, we present another fundamental synchronization construct — the *monitor*. In our presentation of these two constructs, we assume that a process consists of some local data, and a sequential program that can operate on the data. The local data can be accessed by only the sequential program that is encapsulated within the same process. That is, one process cannot directly access the local data of another process. Processes can, however, share global data.

The *critical-region* high-level synchronization construct requires that a variable *v* of type *T*, which is to be shared among many processes, be declared as

var *v*: **shared** *T*;

The variable *v* can be accessed only inside a *region* statement of the following form:

region *v* **when** *B* **do** *S*;

This construct means that, while statement *S* is being executed, no other process can access the variable *v*. The expression *B* is a Boolean expression that governs the access to the critical region. When a process tries to enter the critical-section region, the Boolean expression *B* is evaluated. If the expression is true, statement *S* is executed. If it is false, the process relinquishes the mutual exclusion and is delayed until *B* becomes true and no other process is in the region associated with *v*. Thus, if the two statements,

```
region v when true do S1;
region v when true do S2;
```

are executed concurrently in distinct sequential processes, the result will be equivalent to the sequential execution "*S1* followed by *S2*," or "*S2* followed by *S1*."

The critical-region construct guards against certain simple errors associated with the semaphore solution to the critical-section problem that may be made by a programmer. Note that it does not necessarily eliminate all synchronization errors; rather, it reduces their number. If errors occur in the logic of the program, reproducing a particular sequence of events may not be simple.

The critical-region construct can be effectively used to solve certain general synchronization problems. To illustrate, let us code the bounded-buffer scheme. The buffer space and its pointers are encapsulated in

```
var buffer: shared record
                pool: array [0..n−1] of item;
                count,in,out: integer;
            end;
```

The producer process inserts a new item *nextp* into the shared buffer by executing

```
region buffer when count < n
    do begin
           pool[in] := nextp;
           in := in+1 mod n;
           count := count + 1;
       end;
```

The consumer process removes an item from the shared buffer and puts it in *nextc* by executing

```
region buffer when count > 0
    do begin
           nextc := pool[out];
           out := out+1 mod n;
           count := count − 1;
       end;
```

Let us illustrate how the conditional critical region could be implemented by a compiler. With each shared variable, the following variables are associated:

```
var mutex, first-delay, second-delay: semaphore;
    first-count, second-count: integer;
```

The semaphore *mutex* is initialized to 1; the semaphores *first-delay* and *second-delay* are initialized to 0. The integers *first-count* and *second-count* are initialized to 0.

Mutually exclusive access to the critical section is provided by *mutex*. If a process cannot enter the critical section because the Boolean condition *B* is false, it initially waits on the *first-delay* semaphore. A process waiting on the *first-delay* semaphore is eventually moved to the *second-delay* semaphore before it is allowed to reevaluate its Boolean condition *B*. We keep track of the number of processes waiting on *first-delay* and *second-delay*, with *first-count* and *second-count* respectively.

When a process leaves the critical section, it may have changed the value of some Boolean condition *B* that prevented another process from

```
wait(mutex);
while not B
   do begin
         first-count := first-count + 1;
         if second-count > 0
              then signal(second-delay)
              else signal(mutex);
         wait(first-delay);
         first-count := first-count - 1;
         second-count := second-count + 1;
         if first-count > 0
             then signal(first-delay)
             else signal(second-delay);
         wait(second-delay);
         second-count := second-count - 1;
      end;
S;
if first-count > 0
    then signal(first-delay);
    else if second-count > 0
              then signal(second-delay);
              else  signal(mutex);
```

Figure 6.16 Implementation of the conditional-region construct.

entering the critical section. Accordingly, we must trace through the queue of processes waiting on *first-delay* and *second-delay* (in that order) allowing each process to test its Boolean condition. When a process tests its Boolean condition (during this trace), it may discover that the latter now evaluates to the value *true*. In this case, the process enters its critical section. Otherwise, the process must wait again on the *first-delay* and *second-delay* semaphores, as described previously. Accordingly, for a shared variable *x*, the statement

```
region x when B do S;
```

can be implemented as shown in Figure 6.16. Note that this implementation requires the reevaluation of the expression *B* for any waiting processes every time a process leaves the critical region. If several processes are delayed, waiting for their respective Boolean expressions to become true, this reevaluation overhead may result in inefficient code. There are various optimization methods that we can use to reduce this overhead. Refer to the Bibliographic Notes for relevant references.

6.7 ■ Monitors

Another high-level synchronization construct is the *monitor* type. A monitor is characterized by a set of programmer-defined operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type. The syntax of a monitor is

```
type monitor-name = monitor
   variable declarations

   procedure entry P1 ( ... );
      begin ... end;

   procedure entry P2 ( ... );
      begin ... end;
            .
            .
            .
   procedure entry Pn ( ... );
      begin ... end;

   begin
      initialization code
   end.
```

The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and the formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.

The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly (Figure 6.17). However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the *condition* construct. A programmer who needs to write her own tailor-made synchronization scheme can define one or more variables of type *condition:*

var *x,y*: *condition;*

The only operations that can be invoked on a condition variable are *wait* and *signal*. The operation

x.wait;

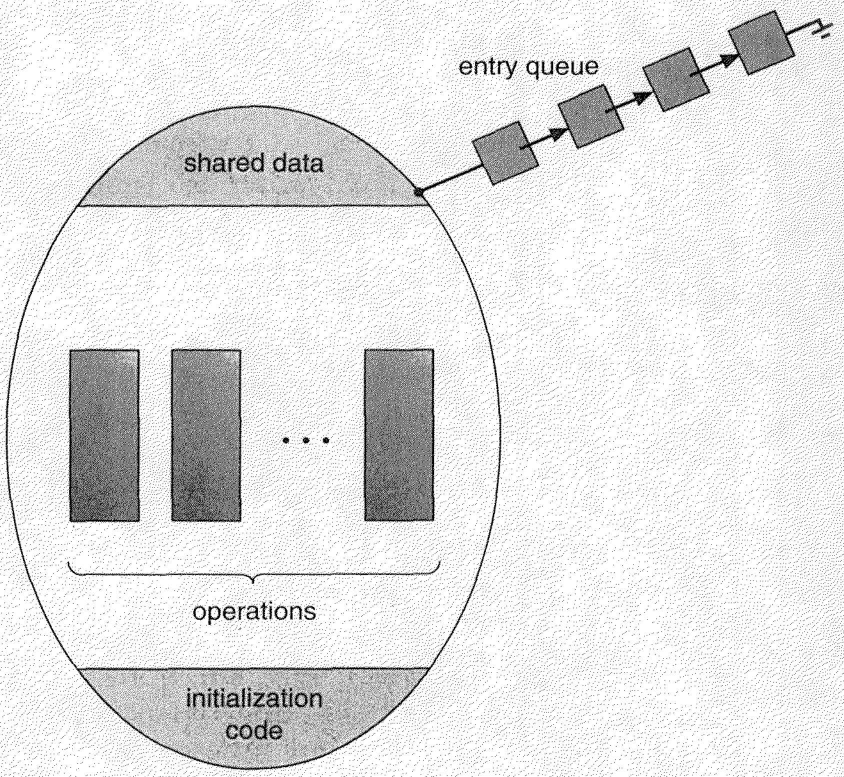


Figure 6.17 Schematic view of a monitor.

means that the process invoking this operation is suspended until another process invokes

x.signal;

The *x.signal* operation resumes exactly one suspended process. If no process is suspended, then the *signal* operation has no effect; that is, the state of *x* is as though the operation was never executed (Figure 6.18). Contrast this operation with the *signal* operation associated with semaphores, which always affects the state of the semaphore.

Now suppose that, when the *x.signal* operation is invoked by a process *P*, there is a suspended process *Q* associated with condition *x*. Clearly, if the suspended process *Q* is allowed to resume its execution, the signaling process *P* must wait. Otherwise, both *P* and *Q* will be active simultaneously within the monitor. Note, however, that both processes can conceptually continue with their execution. Two possibilities exist:

1. *P* either waits until *Q* leaves the monitor, or waits for another condition.

2. *Q* either waits until *P* leaves the monitor, or waits for another condition.

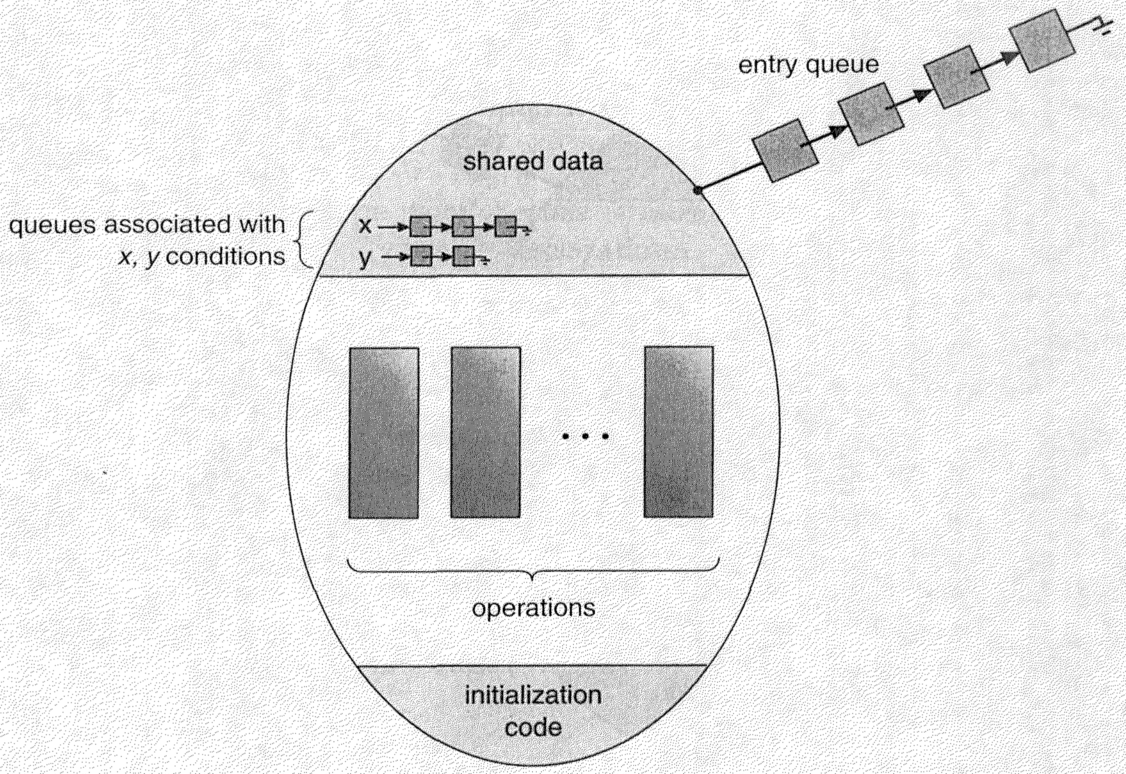


Figure 6.18 Monitor with condition variables.

There are reasonable arguments in favor of adopting either option 1 or option 2. Since P was already executing in the monitor, choice 2 seems more reasonable. However, if we allow process P to continue, the "logical" condition for which Q was waiting may no longer hold by the time Q is resumed.

Choice 1 was advocated by Hoare, mainly because the preceding argument in favor of it translates directly to simpler and more elegant proof rules. A compromise between these two choices was adopted in the language Concurrent Pascal. When process P executes the *signal* operation, it immediately leaves the monitor. Hence, Q is immediately resumed. This model is less powerful than Hoare's, because a process cannot signal more than once during a single procedure call.

Let us illustrate these concepts by presenting a deadlock-free solution to the dining-philosophers problem. Recall that a philosopher is allowed to pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish between three states in which a philosopher may be. For this purpose, we introduce the following data structure:

```
var state: array [0..4] of (thinking, hungry, eating);
```

Philosopher i can set the variable $state[i] = eating$ only if her two neighbors are not eating ($state[i+4 \textbf{ mod } 5] \neq eating$ **and** $state[i+1 \textbf{ mod } 5] \neq eating$).

We also need to declare

```
var self: array [0..4] of condition;
```

where philosopher i can delay herself when she is hungry, but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution. The distribution of the chopsticks is controlled by the monitor shown in Figure 6.19. Philosopher i must invoke the operations *pickup* and *putdown* on an instance *dp* of the *dining-philosophers* monitor in the following sequence:

```
dp.pickup(i);
    ...
   eat
    ...
dp.putdown(i);
```

It is easy to show that this solution ensures that no two neighbors are eating simultaneously, and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death. We shall not present a solution to this problem, but rather shall leave it as an exercise for you.

```
type dining-philosophers = monitor
   var state : array [0..4] of (thinking, hungry, eating);
   var self : array [0..4] of condition;

   procedure entry pickup (i: 0..4);
      begin
         state[i] := hungry;
         test (i);
         if state[i] ≠ eating then self[i].wait;
      end;

   procedure entry putdown (i: 0..4);
      begin
         state[i] := thinking;
         test (i+4 mod 5);
         test (i+1 mod 5);
      end;

   procedure test (k: 0..4);
      begin
         if state[k+4 mod 5] ≠ eating
            and state[k] = hungry
            and state[k+1 mod 5] ≠ eating
            then begin
                    state[k] := eating;
                    self[k].signal;
                 end;
      end;

   begin
      for i := 0 to 4
         do state[i] := thinking;
   end.
```

Figure 6.19 A monitor solution to the dining-philosopher problem.

We shall now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a semaphore *mutex* (initialized to 1) is provided. A process must execute *wait*(*mutex*) before entering the monitor, and must execute *signal*(*mutex*) after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, *next*, is introduced, initialized to

0, on which the signaling processes may suspend themselves. An integer variable *next-count* will also be provided to count the number of processes suspended on *next*. Thus, each external procedure *F* will be replaced by

```
wait(mutex);
   ...
  body of F;
   ...
if next-count > 0
  then signal(next)
  else signal(mutex);
```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented. For each condition *x*, we introduce a semaphore *x-sem* and an integer variable *x-count*, both initialized to 0. The operation *x.wait* can now be implemented as

```
x-count := x-count + 1;
if next-count > 0
  then signal(next)
  else signal(mutex);
wait(x-sem);
x-count := x-count - 1;
```

The operation *x.signal* can be implemented as

```
if x-count > 0
  then begin
          next-count := next-count + 1;
          signal(x-sem);
          wait(next);
          next-count := next-count - 1;
        end;
```

This implementation is applicable to the definitions of monitors given by both Hoare and Brinch Hansen. In some cases, however, the generality of the implementation is unnecessary, and a significant improvement in efficiency is possible. We leave this problem to you in Exercise 6.12.

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition *x*, and an *x.signal* operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use an FCFS ordering, so that the process waiting the longest is resumed first. There are, however, many circumstances in which such a simple scheduling scheme is not adequate. For this purpose, the *conditional-wait* construct can be used; it has the form

x.wait(c);

where *c* is an integer expression that is evaluated when the wait operation is executed. The value of *c*, which is called a *priority number*, is then stored with the name of the process that is suspended. When *x.signal* is executed, the process with the smallest associated priority number is resumed next.

To illustrate this new mechanism, we consider the monitor shown in Figure 6.20, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of its resources, specifies the maximum time it plans to use the resource. The monitor allocates the resource to that process that has the shortest time-allocation request.

A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
    ...
    access the resource;
    ...
R.release;
```

where *R* is an instance of type *resource-allocation*.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequences will be observed. In particular,

- A process might access the resource without first gaining access permission to that resource.
- A process might never release the resource once it has been granted access to that resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing that resource).

Note that the same difficulties are encountered with the critical section construct, and these difficulties are similar in nature to those that encouraged us to develop the critical-region and monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores. Now, we have to worry about the correct use of higher-level

```
type resource-allocation = monitor
    var busy: boolean;
        x: condition;

    procedure entry acquire (time: integer);
        begin
            if busy then x.wait(time);
            busy := true;
        end;

    procedure entry release;
        begin
            busy := false;
            x.signal;
        end;

    begin
        busy := false;
    end.
```

Figure 6.20 A monitor to allocate a single resource.

programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the above problem is to include the resource-access operations within *resource-allocation* monitor. However, this solution will result in scheduling being done according to the built-in monitor-scheduling algorithm, rather than by the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the *resource-allocation* monitor and its managed resource. There are two conditions that we must check to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur, and that the scheduling algorithm will not be defeated.

Although this inspection may be possible for a small, static system, it is not reasonable for a large system or for a dynamic system. This *access-control problem* can be solved only by additional mechanisms that will be elaborated in Chapter 13.

6.8 ■ Synchronization in Solaris 2

To solidify this discussion, we now return to Solaris 2. Before the advent of Solaris 2, SunOS used critical sections to guard important data structures. The system implemented the critical sections by setting the interrupt level to as high as or higher than any interrupt that could modify the same data. Thus, no interrupt would occur that would allow a change to the same data.

In Section 5.5, we described the changes needed to support real-time computing on a time-sharing system. Solaris 2 was designed to provide real-time capabilities, be multithreaded, and support multiprocessors. Continuing to use critical sections would have caused a large performance degradation, as the kernel bottlenecked waiting for entry into critical sections. Further, critical sections could not have been implemented via interrupt elevation because interrupts could occur on other processors on a multiprocessor system. To avoid these problems, Solaris 2 uses *adaptive mutexes* to protect access to every critical data item.

On a multiprocessor system, an adaptive mutex starts as a standard semaphore implemented as a spinlock. If the data are locked, and therefore already in use, the adaptive mutex does one of two things. If the lock is held by a thread that is currently running, the thread waits for the lock to become available because the thread holding the lock is likely to be done soon. If the thread holding the lock is not currently in run state, the thread blocks, going to sleep until it is awakened by the lock being released. It is put to sleep so that it will avoid spinning when the lock will not be freed reasonably quickly. A lock held by a sleeping thread is likely to be in this category. On a uniprocessor system, the thread holding the lock is never running if the lock is being tested by another thread, because only one thread can run at a time. Therefore, on a uniprocessor system, threads always sleep rather than spin if they encounter a lock.

For more complex synchronization situations, Solaris 2 uses condition variables and readers–writers locks. The adaptive mutex method described above is used to protect only those data that are accessed by short code segments. That is, a mutex is used if a lock will be held for less than a few hundred instructions. If the code segment is longer than that, spin waiting will be exceedingly inefficient. For longer code segments, condition variables are used. If the desired lock is already held, the thread issues a wait and sleeps. When a thread frees the lock, it issues a signal to the next sleeping thread in the queue. The extra cost of putting a thread to sleep and waking it, and of the associated context switches, is less than the cost of wasting several hundred instructions waiting in a spinlock.

The readers–writers locks are used to protect data that are accessed frequently, but usually only in a read-only manner. In these circumstances, readers–writers locks are more efficient than are semaphores, because multiple threads may be reading data concurrently, whereas semaphores would always serialize access to the data. Readers–writers locks are expensive to implement, so again they are used on only long sections of code.

6.9 ■ Atomic Transactions

The mutual exclusion of critical sections ensures that the critical sections are executed atomically. That is, if two critical sections are executed concurrently, the result is equivalent to their sequential execution in some unknown order. Although this property is useful in many application domains, there are many cases where we would like to make sure that a critical section forms a single logical unit of work that either is performed in its entirety or is not performed at all. An example is funds transfer, in which one account is debited and another is credited. Clearly, it is essential for data consistency that either both the credit and debit occur, or that neither occur.

The remainder of this section is related to the field of database systems. *Databases* are concerned with the storage and retrieval of data, and with the consistency of the data. Recently, there has been an upsurge of interest in using database-systems techniques in operating systems. Operating systems can be viewed as manipulators of data; as such, they can benefit from the advanced techniques and models available from database research. For instance, many of the ad hoc techniques used in operating systems to manage files could be more flexible and powerful if more formal database methods were used in their place. In Sections 6.9.2 to 6.9.4, we describe what these database techniques are, and how they can be used by operating systems.

6.9.1 System Model

A collection of instructions (operations) that performs a single logical function is called a *transaction*. A major issue in processing transactions is the preservation of atomicity despite the possibility of failures within the computer system. In Section 6.9, we describe various mechanisms for ensuring transaction atomicity. We do so by first considering an environment where only one transaction can be executing at a time. Then, we consider the case where multiple transactions are active simultaneously.

A transaction is a program unit that accesses and possibly updates various data items that may reside on the disk within some files. From our point of view, a transaction is simply a sequence of **read** and **write** operations, terminated by either a **commit** operation or an **abort** operation.

A commit operation signifies that the transaction has terminated its execution successfully, whereas an abort operation signifies that the transaction had to cease its normal execution due to some logical error. A terminated transaction that has completed its execution successfully is *committed;* otherwise, it is *aborted*. The effect of a committed transaction cannot be undone by abortion of the transaction.

A transaction may also cease its normal execution due to a system failure. In either case, since an aborted transaction may have already modified the various data that it has accessed, the state of these data may not be the same as it would be had the transaction executed atomically. So that the atomicity property is ensured, an aborted transaction must have no effect on the state of the data that it has already modified. Thus, the state of the data accessed by an aborted transaction must be restored to what it was just before the transaction started executing. We say that such a transaction has been *rolled back*. It is part of the responsibility of the system to ensure this property.

To determine how the system should ensure atomicity, we need first to identify the properties of devices used for storing the various data accessed by the transactions. Various types of storage media are distinguished by their relative speed, capacity, and resilience to failure.

- **Volatile storage:** Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself and because it is possible to access directly any data item in volatile storage.

- **Nonvolatile storage:** Information residing in nonvolatile storage usually survives system crashes. Examples of media for such storage are disk and magnetic tapes. Disks are more reliable than is main memory, but are less reliable than are magnetic tapes. Both disks and tapes, however, are subject to failure, which may result in loss of information. Currently, nonvolatile storage is slower than volatile storage by several orders of magnitude, because disk and tape devices are electromechanical and require physical motion to access data.

- **Stable storage:** Information residing in stable storage is *never* lost (*never* should be taken with a grain of salt, since theoretically such absolutes cannot be guaranteed). To implement an approximation of such storage, we need to replicate information in several nonvolatile storage caches (usually disk) with independent failure modes, and to update the information in a controlled manner (see Section 12.6).

Here, we are concerned only with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.

6.9.2 Log-Based Recovery

One way to ensure atomicity is to record, on stable storage, information describing all the modifications made by the transaction to the various data it accessed. The most widely used method for achieving this form of recording is *write-ahead logging*. The system maintains, on stable storage, a data structure called the *log*. Each log record describes a single operation of a transaction write, and has the following fields:

- **Transaction name:** The unique name of the transaction that performed the **write** operation
- **Data item name:** The unique name of the data item written
- **Old value:** The value of the data item prior to the write
- **New value:** The value that the data item will have after the write

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.

Before a transaction T_i starts its execution, the record <T_i **starts**> is written to the log. During its execution, any **write** operation by T_i is *preceded* by the writing of the appropriate new record to the log. When T_i commits, the record <T_i **commits**> is written to the log.

Because the information in the log is used in reconstructing the state of the data items accessed by the various transactions, we cannot allow the actual update to a data item to take place before the corresponding log record is written out to stable storage. We therefore require that, prior to a **write**(X) operation being executed, the log records corresponding to X be written onto stable storage.

Note the performance penalty inherent in this system. Two physical writes are required for every logical write requested. Also, more storage is needed: for the data themselves and for the log of the changes. In cases where the data are extremely important, and fast failure recovery is necessary, the price is worth the functionality.

Using the log, the system can handle any failure that does not result in the loss of information on nonvolatile storage. The recovery algorithm uses two procedures:

- **undo**(T_i), which restores the value of all data updated by transaction T_i to the old values
- **redo**(T_i), which sets the value of all data updated by transaction T_i to the new values

The set of data updated by T_i and their respective old and new values can be found in the log.

The **undo** and **redo** operations must be idempotent (that is, multiple executions of an operation have the same result as does one execution) to guarantee correct behavior, even if a failure occurs during the recovery process.

If a transaction T_i aborts, then we can restore the state of the data that it has updated by simply executing **undo**(T_i). If a system failure occurs, we restore the state of all updated data by consulting the log to determine which transactions need to be redone and which need to be undone. This classification of transactions is accomplished as follows:

- Transaction T_i needs to be undone if the log contains the record <T_i **starts**>, but does not contain the record <T_i **commits**>.
- Transaction T_i needs to be redone if the log contains both the record <T_i **starts**> and the record <T_i **commits**>.

6.9.3 Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to make these determinations. There are two major drawbacks to this approach:

1. The searching process is time-consuming.
2. Most of the transactions that, according to our algorithm, need to be redone, have already actually updated the data that the log says they need to modify. Although redoing the data modifications will cause no harm (due to idempotency), it will nevertheless cause recovery to take longer.

To reduce these types of overhead, we introduce the concept of *checkpoints*. During execution, the system maintains the write-ahead log. In addition, the system periodically performs checkpoints, which require the following sequence of actions to take place:

1. Output all log records currently residing in volatile storage (usually main memory) onto stable storage.
2. Output all modified data residing in volatile storage to the stable storage.
3. Output a log record <**checkpoint**> onto stable storage.

The presence of a <**checkpoint**> record in the log allows the system to streamline its recovery procedure. Consider a transaction T_i that committed prior to the checkpoint. The <T_i **commits**> record appears in the log before the <**checkpoint**> record. Any modifications made by T_i must have been written to stable storage either prior to the checkpoint, or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a **redo** operation on T_i.

This observation allows us to refine our previous recovery algorithm. After a failure has occurred, the recovery routine examines the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place. It finds such a transaction by searching the log backward to find the first <**checkpoint**> record, and then finding the subsequent <T_i **start**> record.

Once transaction T_i has been identified, the **redo** and **undo** operations need to be applied to only transaction T_i and all transactions T_j that started executing after transaction T_i. Let us denote these transactions by the set T. The remainder of the log can thus be ignored. The recovery operations that are required are as follows:

- For all transactions T_k in T such that the record <T_k **commits**> appears in the log, execute **redo**(T_k).
- For all transactions T_k in T that have no <T_k **commits**> record in the log, execute **undo**(T_k).

6.9.4 Concurrent Atomic Transactions

Because each transaction is atomic, the concurrent execution of transactions must be equivalent to the case where these transactions executed serially in some arbitrary order. This property, called *serializability*, can be maintained by simply executing each transaction within a critical section. That is, all transactions share a common semaphore *mutex*, which is initialized to 1. When a transaction starts executing, its first action is to execute *wait*(*mutex*). After the transaction either commits or aborts, it executes *signal*(*mutex*).

Although this scheme ensures the atomicity of all concurrently executing transactions, it nevertheless is too restrictive. As we shall see, there are many cases where we can allow transactions to overlap their execution, while maintaining serializability. There are a number of different *concurrency-control* algorithms to ensure serializability. These are described below.

6.9.4.1 Serializability

Consider a system with two data items A and B, that are both read and written by two transactions T_0 and T_1. Suppose that these transactions are executed atomically in the order T_0 followed by T_1. This execution

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure 6.21 Schedule 1: A serial schedule in which T_0 is followed by T_1.

sequence, which is called a *schedule,* is represented in Figure 6.21. In schedule 1 of Figure 6.21, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T_0 appearing in the left column and instructions of T_1 appearing in the right column.

A schedule where each transaction is executed atomically is called a *serial schedule.* Each serial schedule consists of a sequence of instructions from various transactions where the instructions belonging to one single transaction appear together in that schedule. Thus, for a set of n transactions, there exist $n!$ different valid serial schedules. Each serial schedule is correct, because it is equivalent to the atomic execution of the various participating transactions, in some arbitrary order.

If we allow the two transactions to overlap their execution, then the resulting schedule is no longer serial. A nonserial schedule does not necessarily imply that the resulting execution is incorrect (that is, is not equivalent to a serial schedule). To see that this is the case, we need to define the notion of *conflicting operations.* Consider a schedule S in which there are two consecutive operations O_i and O_j of transactions T_i and T_j, respectively. We say that O_i and O_j *conflict* if they access the same data item, and at least one of these operations is a **write** operation. To illustrate the concept of conflicting operations, we consider the nonserial schedule 2 of Figure 6.22. The **write**(A) operation of T_0 conflicts with the **read**(A) operation of T_1. However, the **write**(A) operation of T_1 does not conflict with the **read**(B) operation of T_0, because the two operations access different data items.

Let O_i and O_j be consecutive operations of a schedule S. If O_i and O_j are operations of different transactions and O_i and O_j do not conflict, then we can swap the order of O_i and O_j to produce a new schedule S'. We expect S to be equivalent to S', as all operations appear in the same order in both schedules, except for O_i and O_j, whose order does not matter.

Let us illustrate the swapping idea by considering again schedule 2 of Figure 6.22. As the **write**(A) operation of T_1 does not conflict with the

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 6.22 Schedule 2: A concurrent serializable schedule.

read(B) operation of T_0, we can swap these operations to generate an equivalent schedule. Regardless of the initial system state, both schedules produce the same final system state. Continuing with this procedure of swapping nonconflicting operations, we get:

- Swap the **read**(B) operation of T_0 with the **read**(A) operation of T_1.
- Swap the **write**(B) operation of T_0 with the **write**(A) operation of T_1.
- Swap the **write**(B) operation of T_0 with the **read**(A) operation of T_1.

The final result of these swaps, is schedule 1 in Figure 6.21, which is a serial schedule. Thus, we have shown that schedule 2 is equivalent to a serial schedule. This result implies that, regardless of the initial system state, schedule 2 will produce the same final state as will some serial schedule.

If a schedule S can be transformed into a serial schedule S' by a series of swaps of nonconflicting operations, we say that a schedule S is *conflict serializable*. Thus, schedule 2 is conflict serializable, because it can be transformed into the serial schedule 1.

6.9.4.2 Locking Protocol

One way to ensure serializability is to associate with each data item a lock, and to require that each transaction follow a *locking protocol* that governs how locks are acquired and released. There are various modes in which a data item can be locked. In this section, we restrict our attention to two modes:

- **Shared:** If a transaction T_i has obtained a shared-mode lock (denoted by S) on data item Q, then T_i can read this item, but it cannot write Q.

- **Exclusive:** If a transaction T_i has obtained an exclusive-mode lock (denoted by X) on data item Q, then T_i can both read and write Q.

We require that every transaction request a lock in an appropriate mode on data item Q, depending on the type of operations it will perform on Q.

To access a data item Q, transaction T_i must first lock Q in the appropriate mode. If Q is not currently locked, then the lock is granted, and T_i can now access it. However, if the data item Q is currently locked by some other transaction, then T_i may have to wait. More specifically, suppose that T_i requests an exclusive lock on Q. In this case, T_i must wait until the lock on Q is released. If T_i requests a shared lock on Q, then T_i must wait if Q is locked in exclusive mode. Otherwise, it can obtain the lock and access Q. Notice that this scheme is quite similar to the readers–writers algorithm discussed in Section 6.5.2.

A transaction may unlock a data item that it had locked at an earlier point. It must, however, hold a lock on a data item as long as it accesses that item. Moreover, it is not always desirable for a transaction to unlock a data item immediately after its last access of that data item, because serializability may not be ensured.

One protocol that ensures serializability is the *two-phase locking protocol*. This protocol requires that each transaction issue lock and unlock requests in two phases:

- **Growing phase:** A transaction may obtain locks, but may not release any lock.
- **Shrinking phase:** A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and no more lock requests can be issued.

The two-phase locking protocol ensures conflict serializability (see Exercise 6.21). It does not, however, ensure freedom from deadlock. We note that it is possible that, for a set of transactions, there are conflict serializable schedules that cannot be obtained through the two-phase locking protocol. However, to improve performance over two-phase locking, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data.

6.9.4.3 Timestamp-Based Protocols

In the locking protocols described above, the order between every pair of conflicting transactions is determined at execution time by the first lock that they both request and that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a *timestamp-ordering* scheme.

With each transaction T_i in the system, we associate a unique fixed timestamp, denoted by TS(T_i). This timestamp is assigned by the system before the transaction T_i starts execution. If a transaction T_i has been assigned timestamp TS(T_i), and a new transaction T_j enters the system, then TS(T_i) < TS(T_j). There are two simple methods for implementing this scheme:

- Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system. This method will not work for transactions that occur on separate systems or for processors that do not share a clock.
- Use a logical counter as the timestamp; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system. The counter is incremented after a new timestamp is assigned.

The timestamps of the transactions determine the serializability order. Thus, if TS(T_i) < TS(T_j), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j.

To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp**(Q), which denotes the largest timestamp of any transaction that executed **write**(Q) successfully
- **R-timestamp**(Q), which denotes the largest timestamp of any transaction that executed **read**(Q) successfully

These timestamps are updated whenever a new **read**(Q) or **write**(Q) instruction is executed.

The timestamp-ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order. This protocol operates as follows:

- Suppose that transaction T_i issues **read**(Q).
    - ○ If TS(T_i) < W-timestamp(Q), then this state implies that T_i needs to read a value of Q which was already overwritten. Hence, the **read** operation is rejected, and T_i is rolled back.
    - ○ If TS(T_i) ≥ W-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

- Suppose that transaction T_i issues **write**(Q):
  - If TS(T_i) < R-timestamp(Q), then this state implies that the value of Q that T_i is producing was needed previously and T_i assumed that this value would never be produced. Hence, the **write** operation is rejected, and T_i is rolled back.
  - If TS(T_i) < W-timestamp(Q), then this state implies that T_i is attempting to write an obsolete value of Q. Hence, this **write** operation is rejected, and T_i is rolled back.
  - Otherwise, the **write** operation is executed.

A transaction T_i, which is rolled back by the concurrency-control scheme as result of the issuing of either a **read** or **write** operation, is assigned a new timestamp and is restarted.

To illustrate this protocol, we consider schedule 3 of Figure 6.23 with transactions T_2 and T_3. We assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3, TS(T_2) < TS(T_3), and the schedule is possible under the timestamp protocol.

We note that this execution can also be produced by the two-phase locking protocol. There are, however, schedules that are possible under the two-phase locking protocol but are not possible under the timestamp protocol, and vice versa (see Exercise 6.22).

The timestamp-ordering protocol ensures conflict serializability. This capability follows from the fact that conflicting operations are processed in timestamp order. The protocol ensures freedom from deadlock, because no transaction ever waits.

6.10 ■ Summary

Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided. One solution is to ensure that a critical section of code is in use by only one process or thread at a time. Different algorithms exist for solving the critical-section problem, with the assumption that only storage interlock is available.

The main disadvantage of these user coded solutions is that they all require busy waiting. Semaphores overcome this difficulty. Semaphores can be used to solve various synchronization problems, and can be implemented efficiently, especially if hardware support for atomic operations is available.

Various different synchronization problems (such as the bounded-buffer problem, the readers–writers problem, and the dining-philosophers problem) are important mainly because they are examples of a large class

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Figure 6.23 Schedule 3: A schedule possible under the timestamp protocol.

of concurrency-control problems. These problems are used to test nearly every newly proposed synchronization scheme.

The operating system must provide the means to guard against timing errors. Several language constructs have been proposed to deal with these problems. Critical regions can be used to implement mutual-exclusion and arbitrary-synchronization problems safely and efficiently. Monitors provide the synchronization mechanism for sharing abstract data types. A condition variable provides a method for a monitor procedure to block its execution until it is signaled to continue.

Solaris 2 is an example of a modern operating system which implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing. It uses adaptive mutexes for efficiency when protecting data from short code segments. Condition variables and readers–writers locks are used when longer sections of code need access to data.

A *transaction* is a program unit that must be executed atomically; that is, either all the operations associated with it are executed to completion, or none are performed. To ensure atomicity despite system failure, we can use a *write-ahead* log. All updates are recorded on the log, which is kept in stable storage. If a system crash occurs, the information in the log is used in restoring the state of the updated data items, which is accomplished with the use of the **undo** and **redo** operations. To reduce the overhead in searching the log after a system failure has occurred, we can use a *checkpoint* scheme.

When several transactions overlap their execution, the resulting execution may no longer be equivalent to an execution where these transactions executed atomically. To ensure correct execution, we must use a *concurrency-control* scheme to guarantee *serializability*. There are various different concurrency-control schemes that ensure serializability by either delaying an operation or aborting the transaction that issued the operation. The most common ones are *locking protocols* and *timestamp-ordering* schemes.

■ Exercises

6.1 What is the meaning of the term *busy waiting*? What other kinds of waiting are there? Can busy waiting be avoided altogether? Explain your answer.

6.2 Prove that, in the bakery algorithm (Section 6.2.2), the following property holds: If P_i is in its critical section and P_k ($k \neq i$) has already chosen its $number[k] \neq 0$, then $(number[i], i) < (number[k], k)$.

6.3 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
var flag: array [0..1] of boolean;  (* initially false *)
    turn: 0..1;
```

The structure of process P_i (i = 0 or 1), with P_j (j = 1 or 0) being the other process, is shown in Figure 6.24.

Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
repeat

    flag[i] := true;
    while flag[j]
        do if turn = j
            then begin
                    flag[i] := false;
                    while turn = j do no-op;
                    flag[i] := true;
                 end;

        critical section

    turn := j;
    flag[i] := false;

        remainder section

until false;
```

Figure 6.24 The structure of process P_i in Dekker's algorithm.

```
var j: 0..n;
repeat

    repeat
        flag[i] := want-in;
        j := turn;
        while $j \neq i$
            do if $flag[j] \neq idle$
                then j := turn
                else j := j+1 mod n;
        flag[i] := in-cs;
        j := 0;
        while $(j < n)$ and $(j = i$ or $flag[j] \neq in\text{-}cs)$ do j := j+1;
    until $(j \geq n)$ and $(turn = i$ or $flag[turn] = idle)$;
    turn := i;

        critical section

    j := turn+1 mod n;
    while (flag[j] = idle) do j := j+1 mod n;
    turn := j;
    flag[i] := idle;

        remainder section

until false;
```

Figure 6.25 The structure of P_i in Eisenberg and McGuire's algorithm.

6.4 The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of $n - 1$ turns, was presented by Eisenberg and McGuire. The processes share the following variables:

```
var flag: array [0..n−1] of (idle, wa0-in, in-cs);
    turn: 0..n−1;
```

All the elements of *flag* are initially *idle;* the initial value of *turn* is immaterial (between 0 and $n-1$). The structure of process P_i is shown in Figure 6.25.

Prove that the algorithm satisfies all three requirements for the critical-section problem.

6.5 In Section 6.3, we mentioned that disabling interrupts frequently could affect the system's clock. Explain why it could, and how such effects could be minimized.

6.6 Show that, if the *wait* and *signal* operations are not executed atomically, then mutual exclusion may be violated.

6.7 *The Sleeping-Barber Problem.* A barbershop consists of a waiting room with n chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

6.8 *The Cigarette-Smokers Problem.* Consider a system with three *smoker* processes and one *agent* process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper, and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats. Write a program to synchronize the agent and the smokers.

6.9 Demonstrate that monitors, conditional critical regions, and semaphores are all equivalent, insofar as the same types of synchronization problems can be implemented with them.

6.10 Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

6.11 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 6.10 mainly suitable for small portions.

a. Explain why this assertion is true.

b. Design a new scheme that is suitable for larger portions.

6.12 Suppose that the *signal* statement can appear as only the last statement in a monitor procedure. Suggest how the implementation described in Section 6.7 can be simplified.

6.13 Consider a system consisting of processes P_1, P_2, ..., P_n, each of which has a unique priority number. Write a monitor that allocates three identical line printers to these processes, using the priority numbers for deciding the order of allocation.

6.14 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: The sum of all unique numbers associated with all the processes currently accessing the file must be less than n. Write a monitor to coordinate access to the file.

6.15 Suppose that we replace the *wait* and *signal* operations of monitors with a single construct *await*(B), where B is a general Boolean expression that causes the process executing it to wait until B becomes true.

a. Write a monitor using this scheme to implement the readers–writers problem.

b. Explain why, in general, this construct cannot be implemented efficiently.

c. What restrictions need to be put on the *await* statement so that it can be implemented efficiently? (Hint: Restrict the generality of B; see Kessels [1977].)

6.16 Write a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock, which invokes a procedure *tick* in your monitor at regular intervals.

6.17 Why does Solaris 2 implement multiple locking mechanisms? Under what circumstances does it use spinlocks, blocking semaphores, conditional variables, and readers–writers locks? Why does it use each mechanism?

6.18 Explain the differences, in terms of cost, among the three storage types: volatile, nonvolatile, and stable.

6.19 Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? How does the frequency of checkpoints affect:

- System performance when no failure occurs?
- The time it takes to recover from a system crash?
- The time it takes to recover from a disk crash?

6.20 Explain the concept of transaction atomicity.

6.21 Show that the two-phase locking protocol ensures conflict serializability.

6.22 Show that there are schedules that are possible under the two-phase locking protocol but are not possible under the timestamp protocol, and vice versa.

Bibliographic Notes

The mutual-exclusion algorithms 1 to 2 for two processes were first discussed in the classical paper by Dijkstra [1965a]. Dekker's algorithm (Exercise 6.3) — the first correct software solution to the two-process mutual-exclusion problem — was developed by the Dutch mathematician T. Dekker. This algorithm also was discussed by Dijkstra [1965a]. A simpler solution to the two-process mutual-exclusion problem has since been presented by Peterson [1981] (algorithm 3).

Dijkstra [1965b] presented the first solution to the mutual-exclusion problem for n processes. This solution, however does not have an upper bound on the amount of time a process must wait before that process is allowed to enter the critical section. Knuth [1966] presented the first algorithm with a bound; his bound was 2^n turns. A refinement of Knuth's algorithm by deBruijn [1967] reduced the waiting time to n^2 turns, after which Eisenberg and McGuire [1972] (Exercise 6.4) succeeded in reducing the time to the lower bound of $n - 1$ turns. The bakery algorithm (algorithm 5) was developed by Lamport [1974]; it also requires $n - 1$ turns, but it is easier to program and to understand. Burns [1978] developed the hardware-solution algorithm that satisfies the bounded waiting requirement.

General discussions concerning the mutual-exclusion problem were offered by Lamport [1986, 1991]. A collection of algorithms for mutual exclusion were given by Raynal [1986].

The semaphore concept was suggested by Dijkstra [1965a]. Patil [1971] examined the question of whether semaphores can solve all possible synchronization problems. Parnas [1975] discussed some of the flaws in Patil's arguments. Kosaraju [1973] followed up on Patil's work to produce a problem that cannot be solved by *wait* and *signal* operations. Lipton [1974] discussed the limitation of various synchronization primitives.

The classic process-coordination problems that we have described are paradigms for a large class of concurrency-control problems. The bounded-buffer problem, the dining-philosophers problem, and the sleeping-barber problem (Exercise 6.7) were suggested by Dijkstra [1965a, 1971]. The cigarette-smokers problem (Exercise 6.8) was developed by Patil [1971]. The readers–writers problem was suggested by Courtois et al. [1971]. The issue of concurrent reading and writing was discussed by Lamport [1977]. The problem of synchronization of independent processes was discussed Lamport [1976].

The critical-region concept was suggested by Hoare [1972] and by Brinch Hansen [1972]. The monitor concept was developed by Brinch Hansen [1973]. A complete description of the monitor was given by Hoare [1974]. Kessels [1977] proposed an extension to the monitor to allow automatic signaling. General discussions concerning concurrent programming were offered by Ben-Ari [1990].

Some details of the locking mechanisms used in Solaris 2 are presented in Khanna et al. [1992], Powell et al. [1991], and especially Eykholt et al. [1992]. Note that the locking mechanisms used by the kernel are implemented for user-level threads as well, so the same types of locks are available inside and outside of the kernel.

The write-ahead log scheme was first introduced in System R [Gray et al. 1981]. The concept of serializability was formulated by Eswaran et al. [1976] in connection with their work on concurrency control for System R. The two-phase locking protocol was introduced by Eswaran et al. [1976]. The timestamp-based concurrency-control scheme was provided by Reed [1983]. An exposition of various timestamp-based concurrency-control algorithms was presented by Bernstein and Goodman [1980].

CHAPTER 7

DEADLOCKS

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other waiting processes. This situation is called a *deadlock.* We have already discussed this issue briefly in Chapter 6, in connection with semaphores.

Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in this century. It said, in part: "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

In this chapter, we describe methods that an operating system can use to deal with the deadlock problem. Note, however, that most current operating systems do not provide deadlock-prevention facilities. Such features probably will be added over time, as deadlock problems become more common. Several trends will cause this situation, including larger numbers of processes, many more resources (including CPUs) within a system, and the emphasis on long-lived file and database servers rather than batch systems.

7.1 ■ System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each of which consists of some number of identical instances.

Memory space, CPU cycles, files, and I/O devices (such as printers and tape drives) are examples of resource types. If a system has two CPUs, then the resource type *CPU* has two instances. Similarly, the resource type *printer* may have five instances.

If a process requests an instance of a resource type, the allocation of *any* instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly. For example, a system may have two printers. These two printers may be defined to be in the same resource class if no one cares which printer prints which output. However, if one printer is on the ninth floor and the other is in the basement, then people on the ninth floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer.

A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system only has two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** If the request cannot be granted immediately (for example, the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. **Release:** The process releases the resource.

The request and release of resources are system calls, as explained in Chapter 3. Examples are the **request** and **release device, open** and **close file,** and **allocate** and **free memory** system calls. Request and release of other resources can be accomplished through the *wait* and *signal* operations on semaphores. Therefore, for each use, the operating system checks to make sure that the using process has requested and been allocated the resource. A system table records whether each resource is free or allocated, and, if a resource is allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused by only another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors). However, other types of events may result in deadlocks (for example, the IPC facility discussed in Chapter 4).

To illustrate a deadlock state, we consider a system with three tape drives. Suppose that there are three processes, each holding one of these tape drives. If each process now requests another tape drive, the three processes will be in a deadlock state. Each is waiting for the event "tape drive is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving processes competing for the same resource type.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one tape drive. Suppose that process P_i is holding the tape drive and process P_j is holding the printer. If P_i requests the printer and P_j requests the tape drive, a deadlock occurs.

7.2 ■ Deadlock Characterization

It should be obvious that deadlocks are undesirable. In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from ever starting. Before we discuss the various methods for dealing with the deadlock problem, we shall describe features that characterize deadlocks.

7.2.1 Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion:** At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and wait:** There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

3. **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. **Circular wait:** There must exist a set $\{P_0, P_1, ..., P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent. We shall see in Section 7.4, however, that it is useful to consider each condition separately.

7.2.2 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a *system resource-allocation graph.* This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two types $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system, and $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that process P_i requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge $P_i \rightarrow R_j$ is called a *request edge;* a directed edge $R_j \rightarrow P_i$ is called an *assignment edge.*

Pictorially, we represent each process P_i as a circle, and each resource type R_j as a square. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the square. Note that a request edge points to only the square R_j, whereas an assignment edge must also designate one of the dots in the square.

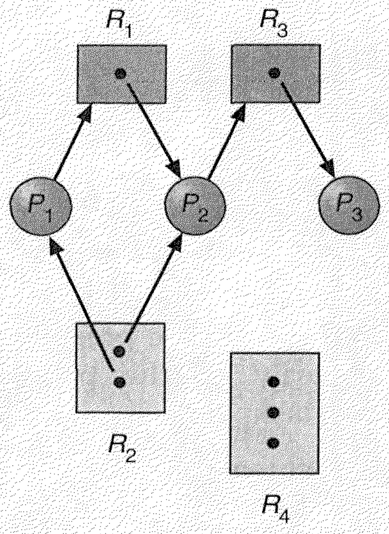


Figure 7.1 Resource-allocation graph.

When process P_i requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process later releases the resource, the assignment edge is deleted.

The resource-allocation graph in Figure 7.1 depicts the following situation.

- The sets P, R, and E:
  - ○ $P = \{P_1, P_2, P_3\}$
  - ○ $R = \{R_1, R_2, R_3, R_4\}$
  - ○ $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Resource instances:
  - ○ One instance of resource type R_1
  - ○ Two instances of resource type R_2
  - ○ One instance of resource type R_3
  - ○ Three instances of resource type R_4
- Process states:
  - ○ Process P_1 is holding an instance of resource type R_2, and is waiting for an instance of resource type R_1.
  - ○ Process P_2 is holding an instance of R_1 and R_2, and is waiting for an instance of resource type R_3.
  - ○ Process P_3 is holding an instance of R_3.

Given the definition of a resource-allocation graph, it can be shown easily that, if the graph contains no cycles, then no process in the system is deadlocked. If, on the other hand, the graph contains a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, let us return to the resource-allocation graph depicted in Figure 7.1. Suppose that process P_3 requests an instance of resource type R_2. Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph (Figure 7.2). At this point, two minimal cycles exist in the system:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

Processes P_1, P_2, and P_3 are deadlocked. Process P_2 is waiting for the resource R_3, which is held by process P_3. Process P_3, on the other hand, is waiting for either process P_1 or process P_2 to release resource R_2. In addition, process P_1 is waiting for process P_2 to release resource R_1.

Now consider the resource-allocation graph in Figure 7.3. In this example, we also have a cycle

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is *not* in a deadlock state. On the other hand, if there is a cycle, then the system may or may not be in a deadlock state. This observation is important when we deal with the deadlock problem.

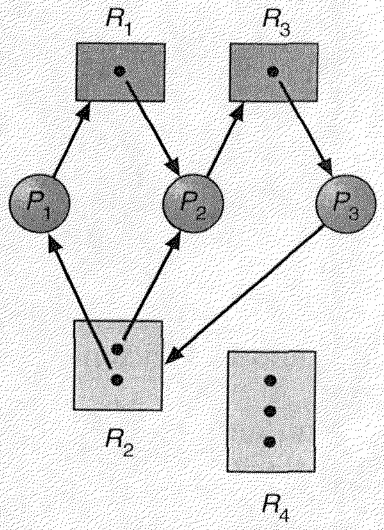


Figure 7.2 Resource-allocation graph with a deadlock.

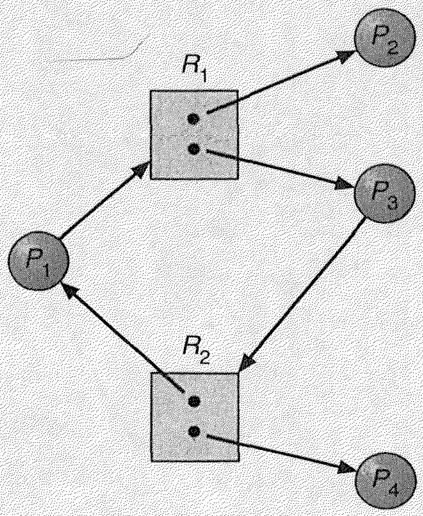


Figure 7.3 Resource-allocation graph with a cycle but no deadlock.

7.3 ■ Methods for Handling Deadlocks

Principally, there are three methods for dealing with the deadlock problem:

- We can use a protocol to ensure that the system will *never* enter a deadlock state.
- We can allow the system to enter a deadlock state and then recover.
- We can ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX.

We shall elaborate briefly on each method. Then, in Sections 7.4 to 7.8, we shall present detailed algorithms.

To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock-avoidance scheme. *Deadlock prevention* is a set of methods for ensuring that at least one of the necessary conditions (Section 7.2.1) cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. We discuss these methods in Section 7.4. *Deadlock avoidance*, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must be delayed. We discuss these schemes in Section 7.5.

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred, and an algorithm to recover from the deadlock (if a deadlock has indeed occurred). We discuss these issues in Section 7.6 and Section 7.7.

If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlock state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in the deterioration of the system performance, because resources are being held by processes that cannot run, and because more and more processes, as they make requests for resources, enter a deadlock state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method does not seem to be a viable approach to the deadlock problem, it is nevertheless used in some operating systems. In many systems, deadlocks occur infrequently (say, once per year); thus, it is cheaper to use this method instead of the costly deadlock prevention, deadlock avoidance, or deadlock detection and recovery methods that must be used constantly. Also, there are circumstances in which the system is in a frozen state without it being in a deadlock state. As an example of this situation, consider a real-time process running at the highest priority (or any process running on a non-preemptive scheduler) and never returning control to the operating system.

7.4 ■ Deadlock Prevention

As we noted in Section 7.2.1, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock. Let us elaborate on this approach by examining each of the four necessary conditions separately.

7.4.1 Mutual Exclusion

The mutual-exclusion condition must hold for nonsharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, it is not possible to prevent deadlocks by denying the mutual-exclusion condition: Some resources are intrinsically nonsharable.

7.4.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a tape drive to a disk file, sorts the disk file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the tape drive, disk file, and the printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the tape drive and disk file. It copies from the tape drive to the disk, then releases both the tape drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

There are two main disadvantages to these protocols. First, *resource utilization* may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.

Second, *starvation* is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

7.4.3 No Preemption

The third necessary condition is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process that is holding some resources requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. That is, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives.

7.4.4 Circular Wait

One way to ensure that the circular-wait condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.

Let $R = \{R_1, R_2, \ldots, R_m\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function $F\colon R \rightarrow N$, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

$$F(\text{tape drive}) = 1,$$
$$F(\text{disk drive}) = 5,$$
$$F(\text{printer}) = 12.$$

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say R_i. After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$. If several instances of the same resource type are needed, a *single* request for all of them must be issued. For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

Alternatively, we can simply require that, whenever a process requests an instance of resource type R_j, it has released any resources R_i such that $F(R_i) \geq F(R_j)$.

If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of processes involved in the circular wait be $\{P_0, P_1, ..., P_n\}$, where P_i is waiting for a resource R_i, which is held by process P_{i+1}. (Modulo arithmetic is used on the indexes, so that P_n is waiting for a resource R_n held by P_0.) Then, since process P_{i+1} is holding resource R_i while requesting resource R_{i+1}, we must have $F(R_i) < F(R_{i+1})$, for all i. But this condition means that $F(R_0) < F(R_1) < ... < F(R_n) < F(R_0)$. By transitivity, $F(R_0) < F(R_0)$, which is impossible. Therefore, there can be no circular wait.

Note that, the function F should be defined according to the normal order of usage of the resources in a system. For example, since the tape drive is usually needed before the printer, it would be reasonable to define F(tape drive) < F(printer).

7.5 ■ Deadlock Avoidance

Deadlock-prevention algorithms, as discussed in Section 7.4, prevent deadlocks by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur, and, hence, that deadlocks cannot hold. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, we might be told that process P will request first the tape drive, and later the printer, before releasing both resources. Process Q, on the other hand, will request first the printer, and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock.

The various algorithms differ in the amount and type of information required. The simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need. Given a priori information, for each process, about the maximum number of resources of each type that may be requested, it is possible to construct an algorithm that ensures that the system will never enter a deadlock state. This algorithm defines the *deadlock-avoidance* approach. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. The resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

7.5.1 Safe State

A state is *safe* if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a *safe sequence*. A sequence of processes $<P_1, P_2, \ldots, P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j < i$. In this situation, if the resources that process P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe*.

A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 7.4). An unsafe state *may* lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlock) states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.

To illustrate, we consider a system with 12 magnetic tape drives and 3 processes: P_0, P_1, and P_2. Process P_0 requires 10 tape drives, process P_1 may need as many as 4, and process P_2 may need up to 9 tape drives. Suppose that, at time t_0, process P_0 is holding 5 tape drives, process P_1 is holding 2, and process P_2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time t_0, the system is in a safe state. The sequence $<P_1, P_0, P_2>$ satisfies the safety condition, since process P_1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives), then process P_0 can get all its tape drives and return them (the system will then have 10 available tape drives), and finally process P_2 could get all its tape drives and return them (the system will then have all 12 tape drives available).

Note that it is possible to go from a safe state to an unsafe state. Suppose that, at time t_1, process P_2 requests and is allocated 1 more tape drive. The system is no longer in a safe state. At this point, only process P_1 can be allocated all its tape drives. When it returns them, the system

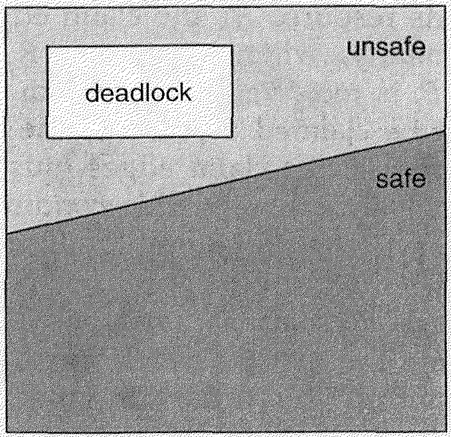


Figure 7.4 Safe, unsafe, and deadlock state spaces.

will have only 4 available tape drives. Since process P_0 is allocated 5 tape drives, but has a maximum of 10, it may then request 5 more tape drives. Since they are unavailable, process P_0 must wait. Similarly, process P_2 may request an additional 6 tape drives and have to wait, resulting in a deadlock.

Our mistake is in granting the request from process P_2 for one more tape drive. If we had made P_2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock situation.

Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

Note that, in this scheme, if a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would be without a deadlock-avoidance algorithm.

7.5.2 Resource-Allocation Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, a variant of the resource-allocation graph defined in Section 7.2.2 can be used for deadlock avoidance.

In addition to the request and assignment edges, we introduce a new type of edge, called a *claim edge*. A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. This edge resembles a request edge in direction, but is represented by a dashed line.

When process P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by P_i, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$. We note that the resources must be claimed a priori in the system. That is, before process P_i starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $P_i \rightarrow R_j$ to be added to the graph only if all the edges associated with process P_i are claim edges.

Suppose that process P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. Note that we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. Therefore, process P_i will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 7.5. Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph (Figure 7.6). A cycle indicates that the system is in an unsafe state. If P_1 requests R_2, and P_2 requests R_1, then a deadlock will occur.

7.5.3 Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the *banker's algorithm*. The name was

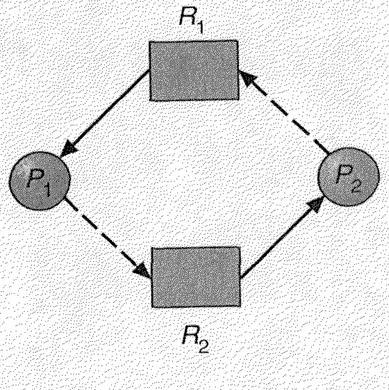


Figure 7.5 Resource-allocation graph for deadlock avoidance.

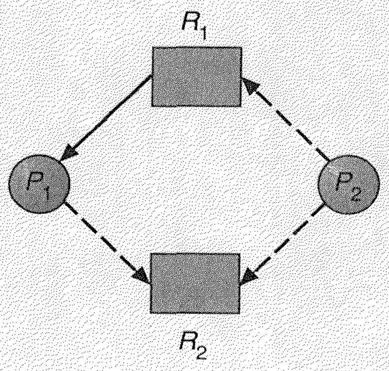


Figure 7.6 An unsafe state in a resource-allocation graph.

chosen because this algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let n be the number of processes in the system and m be the number of resource types. We need the following data structures:

- *Available:* A vector of length m indicates the number of available resources of each type. If $Available[j] = k$, there are k instances of resource type R_j available.

- *Max:* An $n \times m$ matrix defines the maximum demand of each process. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j.

- *Allocation:* An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $Allocation[i,j] = k$, then process P_i is currently allocated k instances of resource type R_j.

- *Need:* An $n \times m$ matrix indicates the remaining resource need of each process. If $Need[i,j] = k$, then process P_i may need k more instances of resource type R_j to complete its task. Note that $Need[i,j] = Max[i,j] - Allocation[i,j]$.

These data structures vary over time in both size and value.

To simplify the presentation of the banker's algorithm, let us establish some notation. Let X and Y be vectors of length n. We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, \ldots, n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices *Allocation* and *Need* as vectors and refer to them as $Allocation_i$ and $Need_i$, respectively. The vector $Allocation_i$ specifies the resources currently allocated to process P_i; the vector $Need_i$ specifies the additional resources that process P_i may still request to complete its task.

7.5.3.1 Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* := *Available* and *Finish*[i] := *false* for $i = 1, 2, \ldots, n$.

2. Find an i such that both

   a. *Finish*[i] = *false*

   b. $Need_i \leq Work$

   If no such i exists, go to step 4.

3. $Work := Work + Allocation_i$
   Finish[i] := *true*
   go to step 2.

4. If *Finish*[i] = *true* for all i, then the system is in a safe state.

This algorithm may require an order of $m \times n^2$ operations to decide whether a state is safe.

7.5.3.2 Resource-Request Algorithm

Let $Request_i$ be the request vector for process P_i. If $Request_i[j] = k$, then process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$$\begin{aligned}&\mathit{Available} := \mathit{Available} - \mathit{Request}_i;\\&\mathit{Allocation}_i := \mathit{Allocation}_i + \mathit{Request}_i;\\&\mathit{Need}_i := \mathit{Need}_i - \mathit{Request}_i;\end{aligned}$$

   If the resulting resource-allocation state is safe, the transaction is completed and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $\mathit{Request}_i$ and the old resource-allocation state is restored.

7.5.3.3 An Illustrative Example

Consider a system with five processes P_0 through P_4 and three resource types A, B, C. Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at time T_0, the following snapshot of the system has been taken:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix *Need* is defined to be *Max* − *Allocation* and is

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies the safety criteria.

Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C, so $\mathit{Request}_1 = (1,0,2)$. To decide whether this request can be immediately granted, we first check that $\mathit{Request}_1 \leq \mathit{Available}$ (that is, $(1,0,2) \leq (3,3,2)$), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies our safety requirement. Hence, we can immediately grant the request of process P_1.

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by P_4 cannot be granted, since the resources are not available. A request for (0,2,0) by P_0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

7.6 ■ Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

In the following discussion, we elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, let us note that a detection and recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm, but also the potential losses inherent in recovering from a deadlock.

7.6.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q. For example, in Figure 7.7, we present a resource-allocation graph and the corresponding wait-for graph.

As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically to *invoke an algorithm* that searches for a cycle in the graph.

An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

7.6.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-detection algorithm that we describe next is applicable to to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm (Section 7.5.3):

- *Available:* A vector of length m indicates the number of available resources of each type.

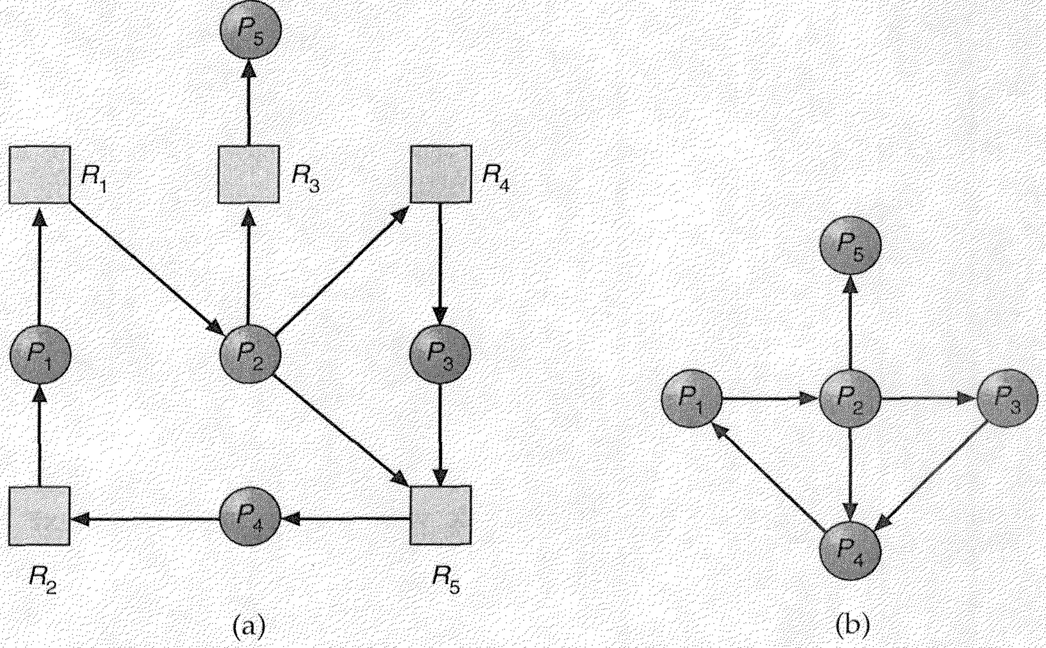


Figure 7.7 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

- *Allocation:* An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- *Request:* An $n \times m$ matrix indicates the current request of each process. If $Request[i,j] = k$, then process P_i is requesting k more instances of resource type R_j.

The less-than relation (<) between two vectors is defined as in Section 7.5.3. To simplify notation, we shall again treat the rows in the matrices *Allocation* and *Request* as vectors, and shall refer to them as $Allocation_i$ and $Request_i$, respectively. The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed. Compare this algorithm with the banker's algorithm of Section 7.5.3.

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work := Available$. For $i = 1, 2, \ldots, n$, if $Allocation_i \neq 0$, then $Finish[i] := false$; otherwise, $Finish[i] := true$.
2. Find an index i such that both

   a. $Finish[i] = false$.

   b. $Request_i \leq Work$.

   If no such i exists, go to step 4.
3. $Work := Work + Allocation_i$
   $Finish[i] := true$
   go to step 2.
4. If $Finish[i]$ = false, for some i, $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if $Finish[i] = false$, then process P_i is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

You may wonder why we reclaim the resources of process P_i (in step 3) as soon as we determine that $Request_i \leq Work$ (in step 2b). We know that P_i is currently *not* involved in a deadlock (since $Request_i \leq Work$). Thus, we take an optimistic attitude, and assume that P_i will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time that the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five processes P_0 through P_4 and three resource types A, B, C. Resource type A has 7 instances, resource type B has 2 instances, and resource type C has 6 instances. Suppose that, at time T_0, we have the following resource-allocation state:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in *Finish*[*i*] = true for all *i*.

Suppose now that process P_2 makes one additional request for an instance of type C. The *Request* matrix is modified as follows:

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process P_0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

7.6.3 Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks can come into being only when some process makes a request that cannot be granted immediately. It is possible that this request is the final request that completes a chain of waiting processes. In the extreme, we could invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the set of processes that is deadlocked, but also the specific process that "caused" the deadlock. (In reality, each of the deadlocked processes is a link in the cycle in the resource graph, so all of them, jointly, caused the deadlock.) If there are many different resource types, one request may cause many cycles in the resource graph, each cycle completed by the most recent request, and "caused" by the one identifiable process.

Of course, invoking the deadlock-detection algorithm for every request may incur a considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at less frequent intervals — for example, once per hour, or whenever CPU utilization drops below 40 percent. (A deadlock eventually cripples system throughput and will cause CPU utilization to drop.) If the detection algorithm is invoked at arbitrary points in time, there may be many cycles in the resource graph. We would generally not be able to tell which of the many deadlocked processes "caused" the deadlock.

7.7 ■ Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system *recover* from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

7.7.1 Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at a great expense, since these processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed later.

- **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Notice that aborting a process may not be easy. If the process was in the midst of updating a file, terminating it in the middle will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on the printer, the system must reset the state of the printer to a correct state before proceeding with the printing of the next job.

If the partial termination method is used, then, given a set of deadlocked processes, we must determine which process (or processes) should be terminated in an attempt to break the deadlock. This determination is a policy decision, similar to CPU-scheduling problems. The question is basically an economic one; we should abort those processes the termination of which will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed, and how much longer the process will compute before completing its designated task
3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

7.7.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim:** Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.

2. **Rollback:** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state, and restart it from that state.

   Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. However, it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

3. **Starvation:** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

   In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a *starvation* situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

7.8 ■ Combined Approach to Deadlock Handling

Researchers have argued that none of the basic approaches for handling deadlocks (prevention, avoidance, and detection) alone is appropriate for the entire spectrum of resource-allocation problems encountered in operating systems. One possibility is to combine the three basic approaches, allowing the use of the optimal approach for each class of resources in the system. The proposed method is based on the notion that resources can be partitioned into classes that are hierarchically ordered. A resource-ordering technique (Section 7.4.4) is applied to the classes. Within each class, the most appropriate technique for handling deadlocks can be used.

It is easy to show that a system that employs this strategy will not be subjected to deadlocks. Indeed, a deadlock cannot involve more than one class, since the resource-ordering technique is used. Within each class, one of the basic approaches is used. Consequently, the system is not subject to deadlocks.

To illustrate this technique, we consider a system that consists of the following four classes of resources:

- **Internal resources:** Resources used by the system, such as a process control block

- **Central memory:** Memory used by a user job
- **Job resources:** Assignable devices (such as a tape drive) and files
- **Swappable space:** Space for each user job on the backing store

One mixed deadlock solution for this system orders the classes as shown, and uses the following approaches to each class:

- **Internal resources:** Prevention through resource ordering can be used, since run-time choices between pending requests are unnecessary.
- **Central memory:** Prevention through preemption can be used, since a job can always be swapped out, and the central memory can be preempted.
- **Job resources:** Avoidance can be used, since the information needed about resource requirements can be obtained from the job-control cards.
- **Swappable space:** Preallocation can be used, since the maximum storage requirements are usually known.

This example shows how various basic approaches can be mixed within the framework of resource ordering, to obtain an effective solution to the deadlock problem.

7.9 ■ Summary

A deadlock state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. Principally, there are three methods for dealing with deadlocks:

- Use some protocol to ensure that the system will never enter a deadlock state.
- Allow the system to enter deadlock state and then recover.
- Ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX.

A deadlock situation may occur if and only if four necessary conditions hold simultaneously in the system: mutual exclusion, hold and wait, no preemption, and circular wait. To prevent deadlocks, we ensure that at least one of the necessary conditions never holds.

Another method for avoiding deadlocks that is less stringent than the prevention algorithms is to have a priori information on how each process will be utilizing the resources. The banker's algorithm needs to know the maximum number of each resource class that may be requested by each process. Using this information, we can define a deadlock-avoidance algorithm.

If a system does not employ a protocol to ensure that deadlocks will never occur, then a detection and recovery scheme must be employed. A deadlock-detection algorithm must be invoked to determine whether a deadlock has occurred. If a deadlock is detected, the system must recover either by terminating some of the deadlocked processes, or by preempting resources from some of the deadlocked processes.

In a system that selects victims for rollback primarily on the basis of cost factors, starvation may occur. As a result, the selected process never completes its designated task.

Finally, researchers have argued that none of these basic approaches alone are appropriate for the entire spectrum of resource-allocation problems in operating systems. The basic approaches can be combined, allowing the separate selection of an optimal one for each class of resources in a system.

■ Exercises

7.1 List three examples of deadlocks that are not related to a computer-system environment.

7.2 Is it possible to have a deadlock involving only one single process? Explain your answer.

7.3 People have said that proper spooling would eliminate deadlocks. Certainly, it eliminates from contention card readers, plotters, printers, and so on. It is even possible to spool tapes (called *staging* them), which would leave the resources of CPU time, memory, and disk space. Is it possible to have a deadlock involving these resources? If it is, how could such a deadlock occur? If it is not, why not? What deadlock scheme would seem best to eliminate these deadlocks (if any are possible), or what condition is violated (if they are not possible)?

7.4 Consider the traffic deadlock depicted in Figure 7.8.

a. Show that the four necessary conditions for deadlock indeed hold in this example.

b. State a simple rule that will avoid deadlocks in this system.

7.5 Suppose that a system is in an unsafe state. Show that it is possible for the processes to complete their execution without entering a deadlock state.

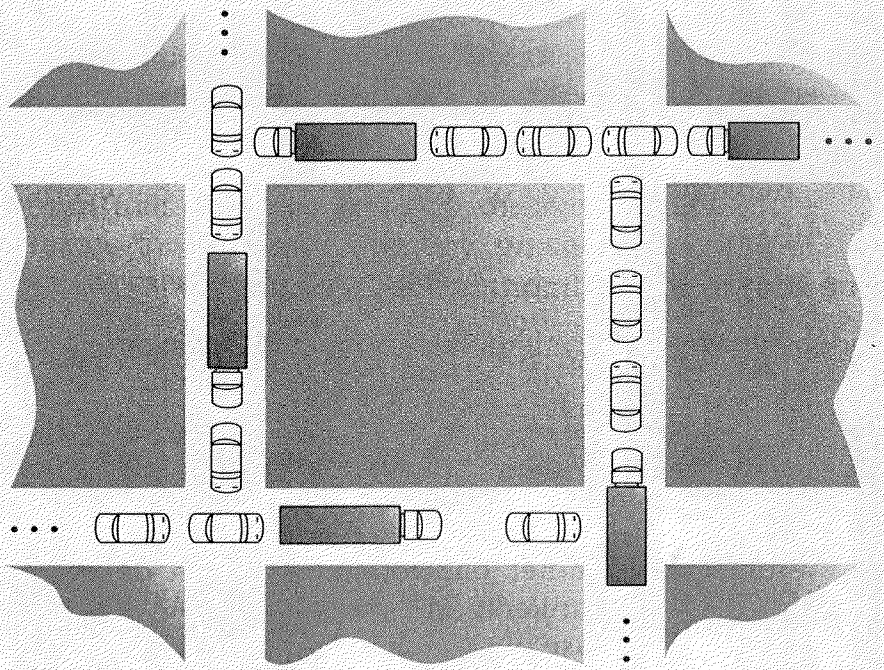


Figure 7.8 Traffic deadlock for Exercise 7.4.

7.6 In a real computer system, neither the resources available nor the demands of processes for resources are consistent over long periods (months). Resources break or are replaced, new processes come and go, new resources are bought and added to the system. If deadlock is controlled by the banker's algorithm, which of the following changes can be made safely (without introducing the possibility of deadlock), and under what circumstances?

a. Increase *Available* (new resources added)

b. Decrease *Available* (resource permanently removed from system)

c. Increase *Max* for one process (the process needs more resources than allowed, it may want more)

d. Decrease *Max* for one process (the process decides it does not need that many resources)

e. Increase the number of processes

f. Decrease the number of processes

7.7 Prove that the safety algorithm presented in Section 7.5.3 requires an order of $m \times n^2$ operations.

7.8 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

7.9 Consider a system consisting of m resources of the same type, being shared by n processes. Resources can be requested and released by processes only one at a time. Show that the system is deadlock free if the following two conditions hold:

a. The maximum need of each process is between 1 and m resources

b. The sum of all maximum needs is less than $m + n$

7.10 Consider a computer system that runs 5000 jobs per month with no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about 10 jobs per deadlock. Each job is worth about $2 (in CPU time), and the jobs terminated tend to be about half-done when they are aborted.

A systems programmer has estimated that a deadlock-avoidance algorithm (like the banker's algorithm) could be installed in the system with an increase in the average execution time per job of about 10 percent. Since the machine currently has 30-percent idle time, all 5000 jobs per month could still be run, although turnaround time would increase by about 20 percent on average.

a. What are the arguments for installing the deadlock-avoidance algorithm?

b. What are the arguments against installing the deadlock-avoidance algorithm?

7.11 Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix *Need*?

b. Is the system in a safe state?

c. If a request from process P_1 arrives for (0,4,2,0) can the request be granted immediately?

7.12 Consider the following resource-allocation policy. Requests and releases for resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked, waiting for resources. If they have the desired resources, then these resources are taken away from them and are given to the requesting process. The vector of resources for which the waiting process is waiting is increased to include the resources that were taken away.

For example, consider a system with three resource types and the vector *Available* initialized to (4,2,2). If process P_0 asks for (2,2,1), it gets them. If P_1 asks for (1,0,1), it gets them. Then, if P_0 asks for (0,0,1), it is blocked (resource not available). If P_2 now asks for (2,0,0), it gets the available one (1,0,0) and one that was allocated to P_0 (since P_0 is blocked). P_0's *Allocation* vector goes down to (1,2,1), and its *Need* vector goes up to (1,0,1).

a. Can deadlock occur? If so, give an example. If not, which necessary condition cannot occur?

b. Can indefinite blocking occur?

7.13 Can a system detect that some of its processes are starving? If you answer "yes," explain how it can. If you answer "no," explain how the system can deal with the starvation problem.

7.14 We can obtain the banker's algorithm for a single resource type from the general banker's algorithm simply by reducing the dimensionality of the various arrays by 1. Show through an example that the multiple-resource-type banker's scheme cannot be implemented by individual application of the single-resource-type scheme to each resource type.

7.15 Suppose that you have coded the deadlock-avoidance safety algorithm and now wish to implement the deadlock-detection algorithm. Can you do so by simply using the safety algorithm code and redefining Max_i = $Waiting_i$ + $Allocation_i$, where $Waiting_i$ is a vector specifying the resources process *i* is waiting for, and $Allocation_i$ is as defined in Section 7.5? Explain your answer.

Bibliographic Notes

Dijkstra [1965a] was one of the first and most influential contributors in the deadlock area. Holt [1972] was the first person to formalize the notion of deadlocks in terms of a graph-theoretical model similar to the one presented in this chapter. The issue of starvation was covered by Holt

[1972]. Hyman [1985] provided the deadlock example from the Kansas legislature.

The various prevention algorithms were suggested by Havender [1968], who has devised the resource-ordering scheme for the IBM OS/360 system.

The banker's algorithm for avoiding deadlocks was developed for a single resource type by Dijkstra [1965a], and was extended to multiple resource types by Habermann [1969]. General discussions concerning avoiding deadlocks by stating claims have been written by Habermann [1969], Holt [1971, 1972], and Parnas and Habermann [1972]. Exercises 7.8 and 7.9 are from Holt [1971].

The deadlock-detection algorithm for multiple instances of a resource type, which was described in Section 7.6.2, was written by Coffman et al. [1971]. The combined approach to deadlocks described in Section 7.8 was originally suggested by Howard [1973].

General surveys and useful bibliographies have been offered by Isloor and Marsland [1980], Newton [1979], and Zoble [1983].

PART THREE

STORAGE MANAGEMENT

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory (at least partially) during execution.

To improve both the utilization of CPU and the speed of its response to its users, the computer must keep several processes in memory. There are many different memory-management schemes. These schemes reflect various approaches to memory management, and the effectiveness of the different algorithms depends on the particular situation. Selection of a memory-management scheme for a specific system depends on many factors, especially on the *hardware* design of the system. Each algorithm requires its own hardware support.

Since main memory is usually too small to accommodate all data and programs permanently, the computer system must provide secondary storage to back up main memory. Most modern computer systems use disks as the primary on-line storage medium for information (both programs and data). The file system provides the mechanism for on-line storage of and access to both data and programs residing on the disks. A file is a collection of related information defined by its creator. Files are mapped, by the operating system, onto physical devices. Files are normally organized into directories to ease their use.

CHAPTER 8

MEMORY MANAGEMENT

In Chapter 5, we showed how the CPU can be shared by a set of processes. As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer's response to its users. To realize this increase in performance, however, we must keep several processes in memory; we must *share* memory.

In this chapter, we discuss various ways to manage memory. The memory-management algorithms vary from a primitive bare-machine approach to paging and segmentation strategies. Each approach has its own advantages and disadvantages. Selection of a memory-management scheme for a specific system depends on many factors, especially on the *hardware* design of the system. As we shall see, many algorithms require hardware support.

8.1 ■ Background

As was shown in Chapter 1, memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

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

8.1.1 Address Binding

Usually, a program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The collection of processes on the disk that are waiting to be brought into memory for execution forms the *input queue.*

The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process does not need to be 00000. This arrangement affects the addresses that the user program can use. In most cases, a user program will go through several steps (some of which may be optional) before being executed (Figure 8.1). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as *count*). A compiler will typically *bind* these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind these relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

Classically, the *binding* of instructions and data to memory addresses can be done at any step along the way:

- **Compile time:** If it is known at compile time where the process will reside in memory, then *absolute* code can be generated. For example, if it is known a priori that a user process resides starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are absolute code bound at compile time.

- **Load time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate *relocatable* code. In this case, final binding is delayed until load time. If the starting address changes, we need only to reload the user code to incorporate this changed value.

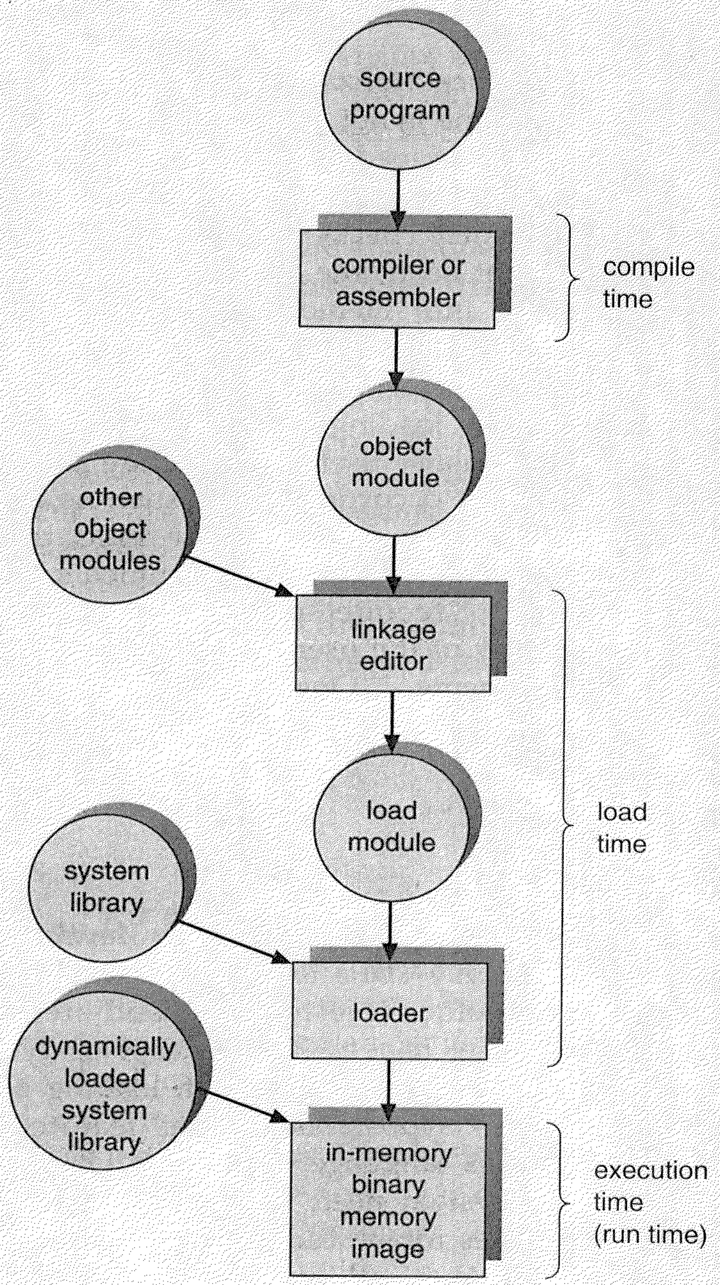


Figure 8.1 Multistep processing of a user program.

- **Execution time:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed in Section 8.2.

A major portion of this chapter is devoted to showing how these various bindings can be implemented effectively in a computer system, and discussing appropriate hardware support.

8.1.2 Dynamic Loading

To obtain better memory-space utilization, we can use *dynamic loading.* With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not been, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then, control is passed to the newly loaded routine.

The advantage of dynamic loading is that an unused routine is never loaded. This scheme is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is actually used (and hence actually loaded) may be much smaller.

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a scheme. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

8.1.3 Dynamic Linking

Notice that Figure 8.1 also shows *dynamically linked* libraries. Most operating systems support only *static linking,* in which system language libraries are treated like any other object module and are combined by the loader into the binary program image. The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, all programs on a system need to have a copy of their language library (or at least the routines referenced by the program) included in the executable image. This requirement wastes both disk space and main memory. With dynamic linking, a *stub* is included in the image for each library-routine reference. This stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine, or how to load the library if the routine is not already present.

When this stub is executed, it checks to see whether the needed routine is already in memory. If the routine is not in memory, the program loads it into memory. Either way, the stub replaces itself with the address of the routine, and executes the routine. Thus, the next time that that code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking. Under this scheme, all processes that use a language library execute only one copy of the library code.

This feature can be extended to library updates (such as bug fixes). A library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Minor changes retain the same version number, whereas major changes increment the version number. Thus, only programs that are compiled with the new library version are affected by the incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library. This system is also known as *shared libraries.*

Unlike dynamic loading, dynamic linking generally requires some help from the operating system. If the processes in memory are protected from one another (Section 8.4.1), then the operating system is the only entity that can check to see whether the needed routine is in another processes' memory space, and can allow multiple processes to access the same memory addresses. This concept is expanded when used in conjunction with paging, as discussed in Section 8.5.5.

8.1.4 Overlays

In our discussion so far, the entire program and data of a process must be in physical memory for the process to execute. The size of a process is limited to the size of physical memory. So that a process can be larger than the amount of memory allocated to it, a technique called *overlays* is sometimes used. The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer needed.

As an example, consider a two-pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code. We may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table, and common support routines used by both pass 1 and pass 2. Assume that the sizes of these components are as follows (K stands for "kilobyte," which is 1024 bytes):

Pass 1	70K
Pass 2	80K
Symbol table	20K
Common routines	30K

To load everything at once, we would require 200K of memory. If only

150K is available, we cannot run our process. However, notice that pass 1 and pass 2 do not need to be in memory at the same time. We thus define two overlays: Overlay *A* is the symbol table, common routines, and pass 1, and overlay *B* is the symbol table, common routines, and pass 2.

We add an overlay driver (10K) and start with overlay *A* in memory. When we finish pass 1, we jump to the overlay driver, which reads overlay *B* into memory, overwriting overlay *A*, and then transfers control to pass 2. Overlay *A* needs only 120K, whereas overlay *B* needs 130K (Figure 8.2). We can now run our assembler in the 150K of memory. It will load somewhat faster because fewer data need to be transferred before execution starts. However, it will run somewhat slower, due to the extra I/O to read the code for overlay *B* over the code for overlay *A*.

The code for overlay *A* and the code for overlay *B* are kept on disk as absolute memory images, and are read by the overlay driver as needed. Special relocation and linking algorithms are needed to construct the overlays.

As in dynamic loading, overlays do not require any special support from the operating system. They can be implemented completely by the user with simple file structures, reading from the files into memory and then jumping to that memory and executing the newly read instructions. The operating system notices only that there is more I/O than usual.

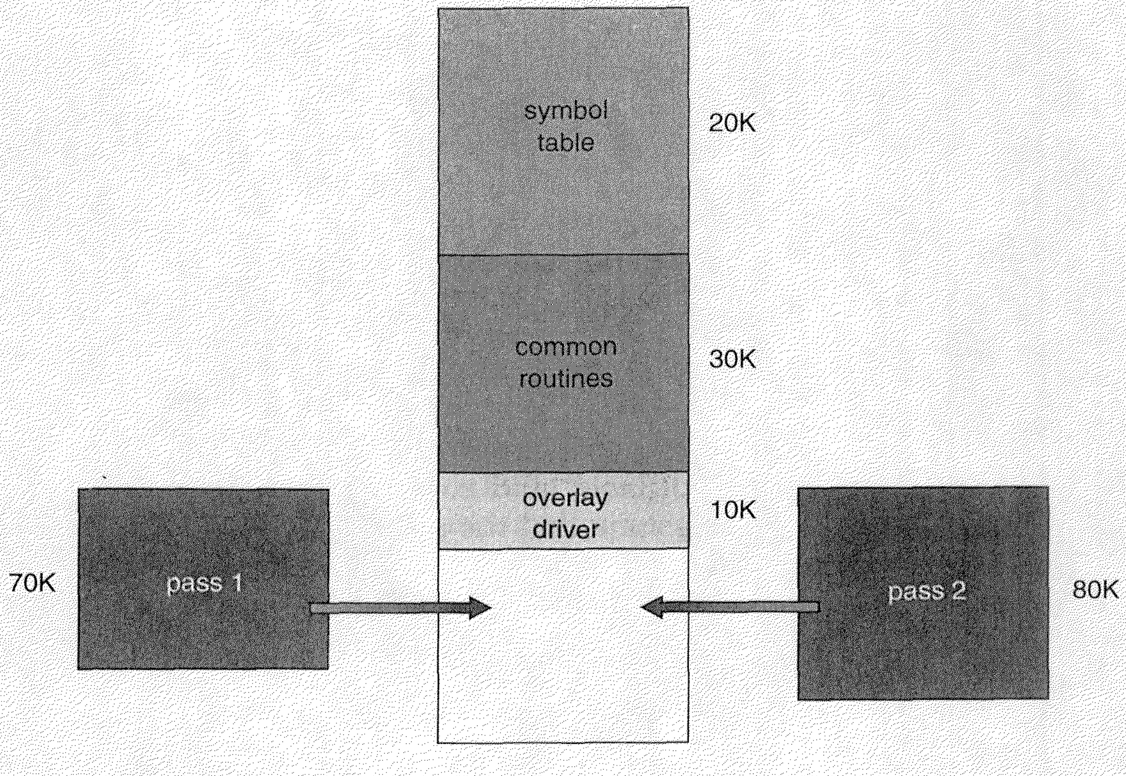


Figure 8.2 Overlays for a two-pass assembler.

The programmer, on the other hand, must design and program the overlay structure properly. This task can be a major undertaking, requiring complete knowledge of the structure of the program, its code, and its data structures. Because the program is, by definition, large (small programs do not need to be overlayed), obtaining a sufficient understanding of the program may be difficult. For these reasons, the use of overlays is currently limited to microcomputer and other systems that have limited amounts of physical memory and that lack hardware support for more advanced techniques. Some microcomputer compilers provide to the programmer support of overlays to make the task easier. Automatic techniques to run large programs in limited amounts of physical memory are certainly preferable.

8.2 ■ Logical versus Physical Address Space

An address generated by the CPU is commonly referred to as a *logical address*, whereas an address seen by the memory unit (that is, the one loaded into the *memory address register* of the memory) is commonly referred to as a *physical address*.

The compile-time and load-time address-binding schemes result in an environment where the logical and physical addresses are the same. However, the execution-time address-binding scheme results in an environment where the logical and physical addresses differ. In this case, we usually refer to the logical address as a *virtual address*. We use *logical address* and *virtual address* interchangeably in this text. The set of all logical addresses generated by a program is referred to as a *logical address space;* the set of all physical addresses corresponding to these logical addresses is referred to as a *physical address space*. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

The run-time mapping from virtual to physical addresses is done by the *memory-management unit (MMU)*, which is a hardware device. There are a number of different schemes for accomplishing such a mapping, as will be discussed in Sections 8.4.1, 8.5, 8.6, and 8.7. For the time being, we shall illustrate this mapping with a simple MMU scheme, which is a generalization of the base-register scheme described in Section 2.4.

As illustrated in Figure 8.3, this scheme requires hardware support slightly different from the hardware configuration discussed in Section 2.4. The base register is now called a *relocation* register. The value in the relocation register is *added* to every address generated by a user process at the time it is sent to memory. For example, if the base is at 14,000, then an attempt by the user to address location 0 is dynamically relocated to location 14,000; an access to location 346 is mapped to location 14346. The MS-DOS operating system running on the Intel 80X86 family of processors uses four relocation registers when loading and running processes.

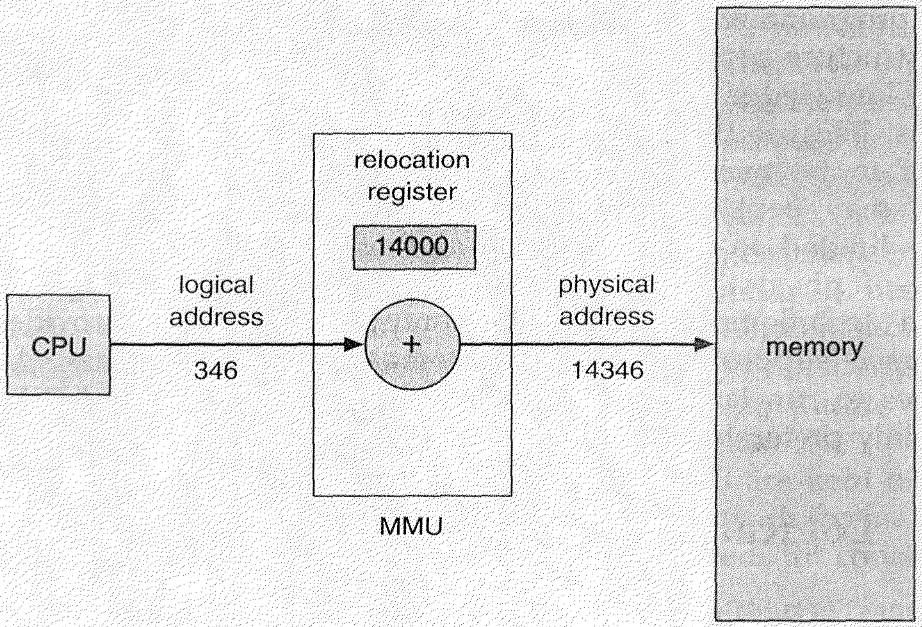


Figure 8.3 Dynamic relocation using a relocation register.

Notice that the user program never sees the *real* physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, compare it to other addresses — all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with *logical* addresses. The memory-mapping hardware converts logical addresses into physical addresses. This form of execution-time binding was discussed in Section 8.1.1. The final location of a referenced memory address is not determined until the reference is made.

Notice also that we now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range $R + 0$ to $R + max$ for a base value R). The user generates only logical addresses and thinks that the process runs in locations 0 to *max*. The user program supplies logical addresses; these logical addresses must be mapped to physical addresses before they are used.

The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management.

8.3 ■ Swapping

A process needs to be in memory to be executed. A process, however, can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed (Figure 8.4). In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that there are always processes in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called *roll out, roll in.*

Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously. This restriction is dictated by the method of address binding. If binding is done at assembly or load time, then the process cannot be moved to different locations. If execution-time binding is being used, then it is possible to swap a process into a different memory space, because the physical addresses are computed during execution time.

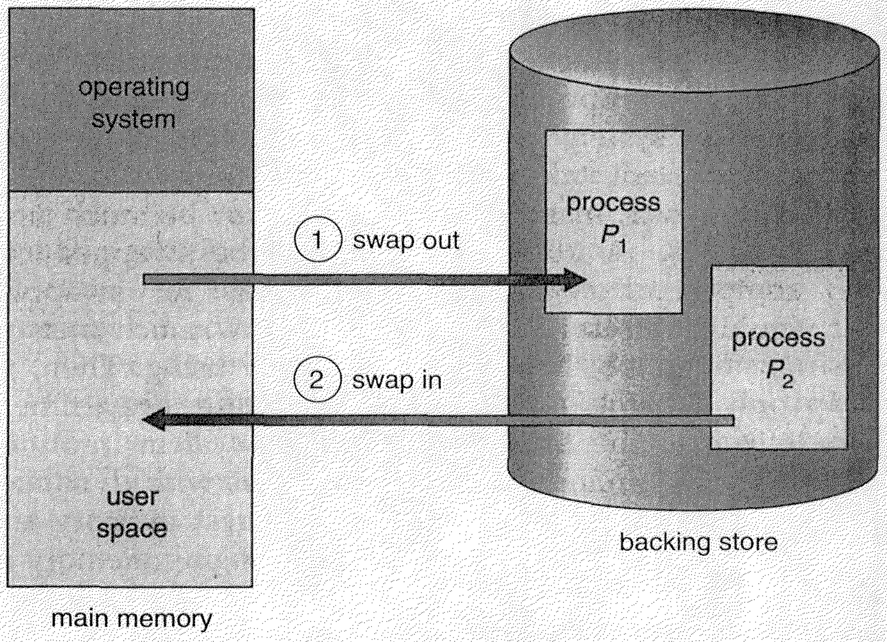


Figure 8.4 Swapping of two processes using a disk as a backing store.

Swapping requires a *backing store*. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and must provide direct access to these memory images. The system maintains a *ready queue* consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If the process is not, and there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers as normal and transfers control to the selected process.

It should be clear that the context-switch time in such a swapping system is fairly high. To get an idea of the context-switch time, let us assume that the user process is of size 100K and the backing store is a standard hard disk with a transfer rate of 1 megabyte per second. The actual transfer of the 100K process to or from memory takes

$$\begin{aligned} \text{100K / 1000K per second} &= \text{1/10 second} \\ &= \text{100 milliseconds} \end{aligned}$$

Assuming that no head seeks are necessary and an average latency of 8 milliseconds, the swap time takes 108 milliseconds. Since we must both swap out and swap in, the total swap time is then about 216 milliseconds.

For efficient CPU utilization, we want our execution time for each process to be long relative to the swap time. Thus, in a round-robin CPU-scheduling algorithm, for example, the time quantum should be substantially larger than 0.216 seconds.

Notice that the major part of the swap time is transfer time. The total transfer time is directly proportional to the *amount* of memory swapped. If we have a computer system with 1 megabyte of main memory and a resident operating system taking 100K, the maximum size of the user process is 900K. However, many user processes may be much smaller than this size — say, 100K. A 100K process could be swapped out in 108 milliseconds, compared to the 908 milliseconds for swapping 900K. Therefore, it would be useful to know exactly how much memory a user process *is* using, not simply how much it *might be* using. Then, we would need to swap only what is actually used, reducing swap time. For this scheme to be effective, the user must keep the system informed of any changes in memory requirements. Thus, a process with dynamic memory requirements will need to issue system calls (**request memory** and **release memory**) to inform the operating system of its changing memory needs.

There are other constraints on swapping. If we want to swap a process, we must be sure that it is completely idle. Of particular concern is any pending I/O. If a process is waiting for an I/O operation, we may want to swap that process to free up its memory. However, if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped. Assume that the I/O operation was queued because the device was busy. Then, if we were to swap out process P_1 and swap in process P_2, the I/O operation might then attempt to use memory that now belongs to process P_2. The two main solutions to this problem are (1) never to swap a process with pending I/O, or (2) to execute I/O operations only into operating-system buffers. Transfers between operating-system and process memory then occur only when the process is swapped in.

The assumption that swapping requires few if any head seeks needs further explanation. We postpone discussing this issue until Chapter 12, where secondary-storage structure is covered. Generally, swap space is allocated as a separate chunk of disk, separate from the file system, so that its use is as fast as possible.

Currently, standard swapping is used in few systems. It requires too much swapping time and provides too little execution time to be a reasonable memory-management solution. Modified versions of swapping, however, are found on many systems.

A modification of swapping is used in many versions of UNIX. Swapping was normally disabled, but would start if many processes were running and were using a threshold amount of memory. Swapping would again be halted if the load on the system was reduced. Memory management in UNIX is described fully in Section 19.6.

PCs lack sophisticated hardware (or operating systems that take advantage of the hardware) to implement more advanced memory-management methods, but they are being used to run multiple, large processes, and a modified version of swapping is the best means to allow them to run. A prime example is the Microsoft Windows operating system, which supports concurrent execution of processes in memory. If a new process is loaded and there is insufficient main memory, an old process is swapped to disk. This operating system, however, does not provide full swapping, because the user, rather than the scheduler, decides when it is time to swap one process for another. Any swapped-out process remains swapped out (and not executing) until the user selects that process to run. The follow-on Microsoft operating system, Windows/NT, takes advantage of advanced MMU features now found even on PCs. In Section 8.7.2, we describe the memory-management hardware found on the Intel 386 family of processors used in many PCs. In that section, we also describe the memory management used on this CPU by another advanced operating system for PCs: IBM OS/2.

8.4 ■ Contiguous Allocation

The main memory must accommodate both the operating system and the various user processes. The memory is usually divided into two partitions,

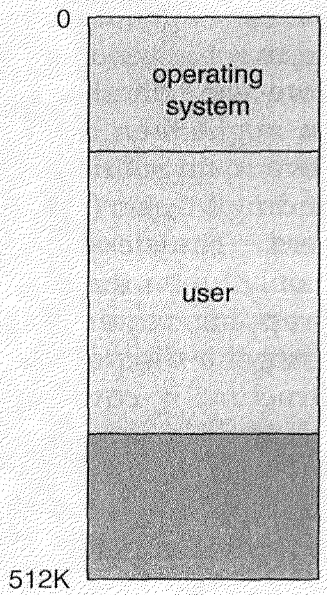


Figure 8.5 Memory partition.

one for the resident operating system, and one for the user processes. It is possible to place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, it is more common to place the operating system in low memory. Thus, we shall discuss only the situation where the operating system resides in low memory (Figure 8.5). The development of the other situation is similar.

8.4.1 Single-Partition Allocation

If the operating system is residing in low memory and the user processes are executing in high memory, we need to protect the operating-system code and data from changes (accidental or malicious) by the user processes. We also need to protect the user processes from one another. We can provide this protection by using a relocation register, as discussed in Section 8.2, with a limit register, as discussed in Section 2.5.3. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100,040 and limit = 74,600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address *dynamically* by adding the value in the relocation register. This mapped address is sent to memory (Figure 8.6).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked

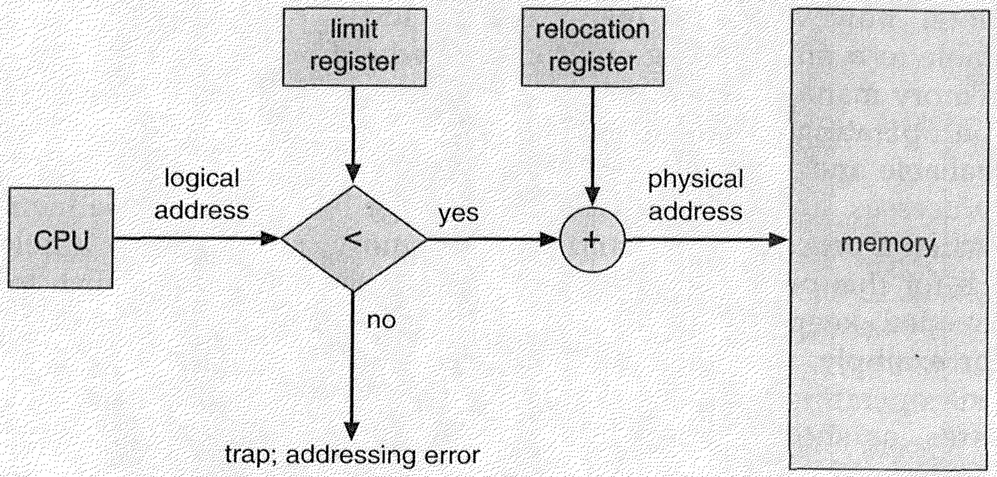


Figure 8.6 Hardware support for relocation and limit registers.

against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

Note that the relocation-register scheme provides an effective way to allow the operating-system size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, it is undesirable to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called *transient* operating-system code; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

8.4.2 Multiple-Partition Allocation

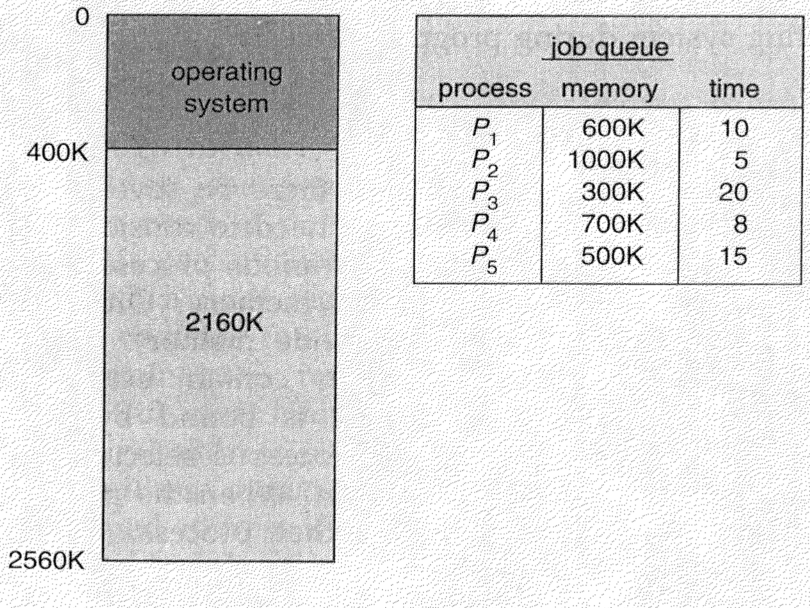
Because it is desirable, in general, that there be several user processes residing in memory at the same time, we need to consider the problem of how to allocate available memory to the various processes that are in the input queue waiting to be brought into memory. One of the simplest schemes for memory allocation is to divide memory into a number of fixed-sized *partitions*. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This scheme was originally used by the IBM OS/360 operating system (called MFT); it is no longer in use. The scheme described next is a generalization of the fixed-partition scheme (called MVT) and is used primarily in a batch environment.

We note, however, that many of the ideas presented here are also applicable to a time-sharing environment where pure segmentation is used for memory management (Section 8.6).

The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes, and is considered as one large block of available memory, a *hole*. When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.

For example, assume that we have 2560K of memory available and a resident operating system of 400K. This situation leaves 2160K for user processes, as shown in Figure 8.7. Given the input queue in the figure, and FCFS job scheduling, we can immediately allocate memory to processes P_1, P_2, and P_3, creating the memory map of Figure 8.8(a). We have a hole of size 260K that cannot be used by any of the remaining processes in the input queue. Using a round-robin CPU-scheduling with a quantum of 1 time unit, process P_2 will terminate at time 14, releasing its memory. This situation is illustrated in Figure 8.8(b). We then return to our job queue and schedule the next process, process P_4, to produce the memory map of Figure 8.8(c). Process P_1 will terminate at time 28 to produce Figure 8.8(d); process P_5 is then scheduled, producing Figure 8.8(e).

This example illustrates the general structure of the allocation scheme. As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which



job queue

process	memory	time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

Figure 8.7 Scheduling example.

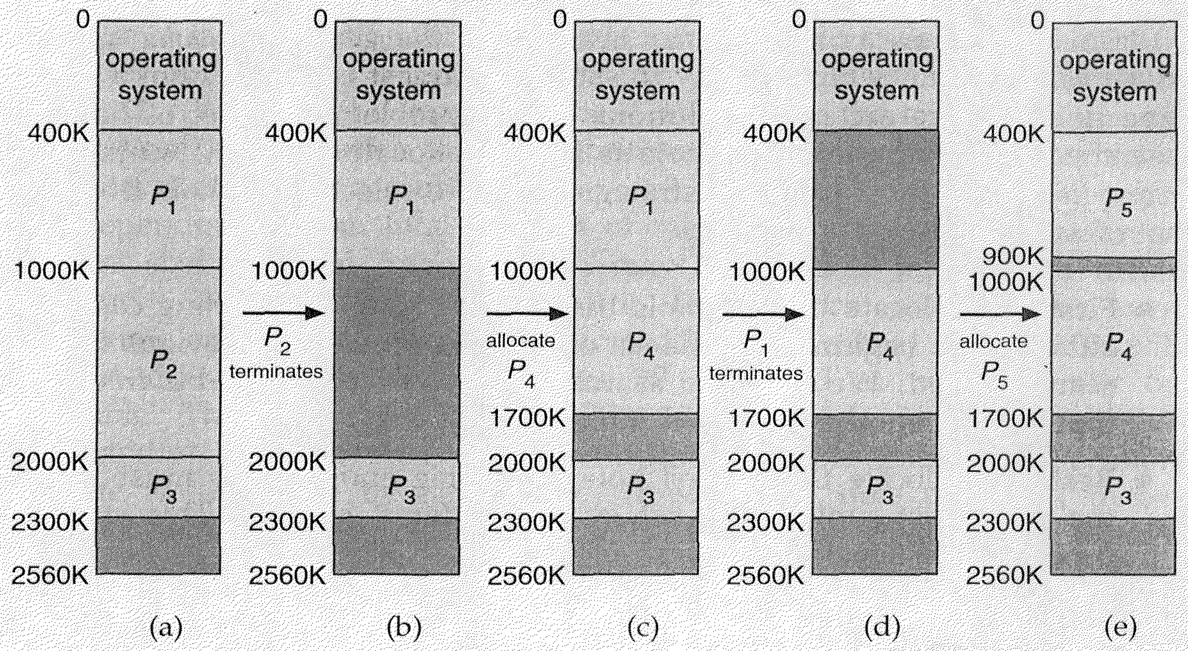


Figure 8.8 Memory allocation and long-term scheduling.

processes are allocated memory. When a process is allocated space, it is loaded into memory and it can then compete for the CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

At any given time, we have a list of available block sizes and the input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied; no available block of memory (hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

In general, there is at any time a *set* of holes, of various sizes, scattered throughout memory. When a process arrives and needs memory, we search this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, we merge these adjacent holes to form one larger hole. At this point, we may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general *dynamic storage-allocation* problem, which is how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate. *First-fit, best-fit,* and *worst-fit* are the most common strategies used to select a free hole from the set of available holes.

- **First-fit:** Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best-fit:** Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy produces the smallest leftover hole.
- **Worst-fit:** Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first-fit and best-fit are better than worst-fit in terms of decreasing both time and storage utilization. Neither first-fit nor best-fit is clearly better in terms of storage utilization, but first-fit is generally faster.

8.4.3 External and Internal Fragmentation

The algorithms described in Section 8.4.2 suffer from *external fragmentation.* As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when enough total memory space exists to satisfy a request, but it is not contiguous; storage is fragmented into a large number of small holes. Looking back at Figure 8.8, we can see two such situations. In Figure 8.8(a), there is a total external fragmentation of 260K, a space that is too small to satisfy the requests of either of the two remaining processes, P_4 and P_5. In Figure 8.8(c), however, we have a total external fragmentation of 560K (= 300K + 260K). This space would be large enough to run process P_5 (which needs 500K), *except* that this free memory is not contiguous. The free memory space is fragmented into two pieces, neither one of which is large enough, by itself, to satisfy the memory request of process P_5.

This fragmentation problem can be severe. In the worst case, we could have a block of free (wasted) memory between every two processes. If all this memory were in one big free block, we might be able to run several more processes. The selection of first-fit versus best-fit can affect the amount of fragmentation. (First-fit is better for some systems, and best-fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece — the one on the top, or the one on the bottom?) No matter which algorithms are used, however, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be either a minor or a major problem. Statistical analysis of first-fit, for instance, reveals that, even with some optimization, given N allocated blocks, another $0.5N$ blocks will be lost due to fragmentation. That is, one-third of memory may be unusable! This property is known as the *50-percent rule.*

Another problem that arises with the multiple partition allocation scheme is illustrated by Figure 8.9. Consider the hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach is to allocate very small holes as part of the larger request. Thus, the allocated memory may be slightly larger than the requested memory. The difference between these two numbers is *internal fragmentation* — memory that is internal to a partition, but is not being used.

One solution to the problem of external fragmentation is *compaction.* The goal is to shuffle the memory contents to place all free memory together in one large block. For example, the memory map of Figure 8.8(e) can be compacted, as shown in Figure 8.10. The three holes of sizes 100K, 300K, and 260K can be compacted into one hole of size 660K.

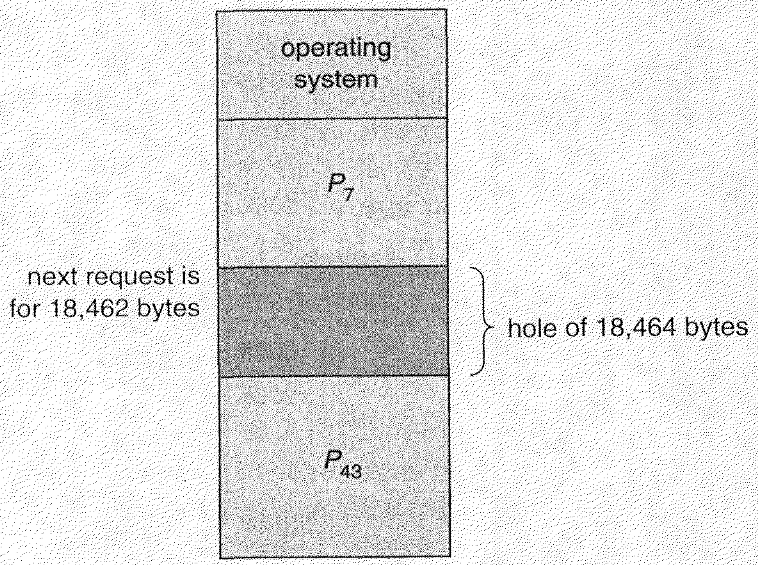


Figure 8.9 Memory allocation made in some multiple of bytes.

Compaction is not always possible. Notice that, in Figure 8.10, we moved processes P_4 and P_3. For these processes to be able to execute in their new locations, all internal addresses must be relocated. If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible *only* if relocation is dynamic, and is done at execution time.

If addresses are relocated dynamically, relocation requires only moving the program and data, and then changing the base register to reflect the new base address.

When compaction is possible, we must determine its cost. The simplest compaction algorithm is simply to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be quite expensive.

Consider the memory allocation shown in Figure 8.11. If we use this simple algorithm, we must move processes P_3 and P_4, for a total of 600K moved. In this situation, we could simply move process P_4 above process P_3, moving only 400K, or move process P_3 below process P_4, moving only 200K. Note that, in this last instance, our one large hole of available memory is not at the end of memory, but rather is in the middle. Also notice that, if the queue contained only one process that wanted 450K, we could satisfy that *particular* request by moving process P_2 somewhere else (such as below process P_4). Although this solution does not create a single large hole, it does create a hole big enough to satisfy the immediate request. Selecting an optimal compaction strategy is quite difficult.

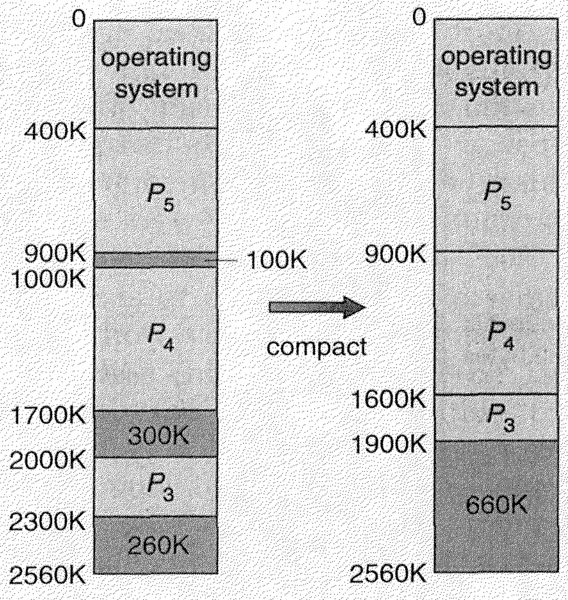


Figure 8.10 Compaction.

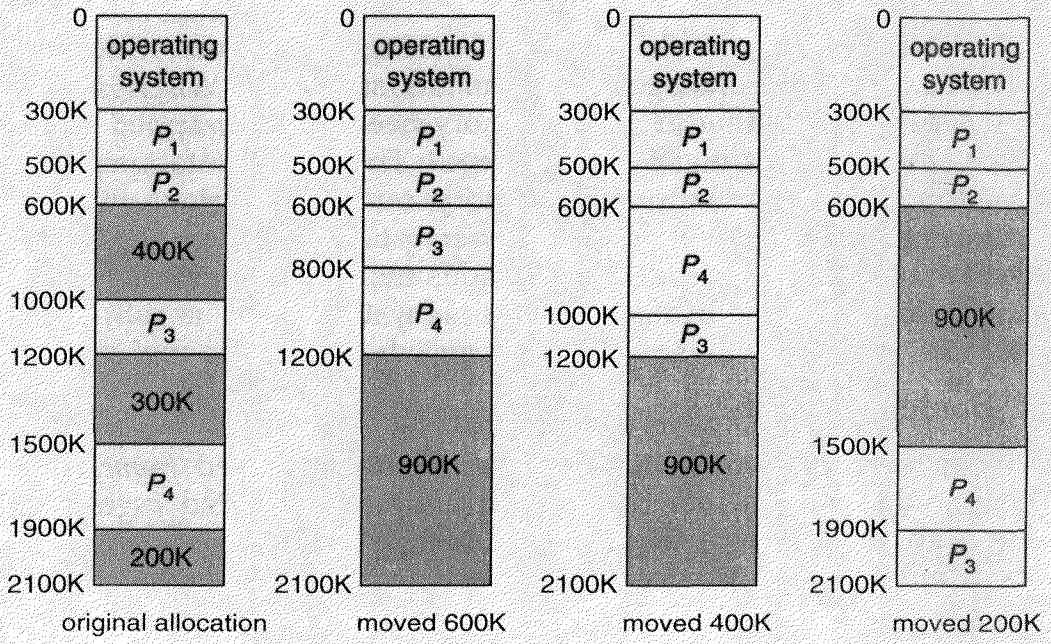


Figure 8.11 Comparison of some different ways to compact memory.

Swapping can also be combined with compaction. A process can be rolled out of main memory to a backing store and rolled in again later. When the process is rolled out, its memory is released, and perhaps is reused for another process. When the process is to be rolled back in, several problems may arise. If static relocation is used, the process must be rolled into the exact same memory locations that it occupied previously. This restriction may require that other processes be rolled out to free that memory.

If dynamic relocation (such as with base and limit registers) is used, then a process can be rolled into a different location. In this case, we find a free block, compacting if necessary, and roll in the process.

One approach to compaction is to roll out those processes to be moved, and to roll them into different memory locations. If swapping or roll-in, roll-out is already a part of the system, the additional code for compaction may be minimal.

8.5 ■ Paging

Another possible solution to the external fragmentation problem is to permit the physical address space of a process to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. One way of implementing this solution is through the use of a *paging* scheme. Paging avoids the considerable problem of fitting the varying-sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered. When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The fragmentation problems discussed in connection with main memory are also prevalent with backing store, except that access is much slower, so compaction is impossible. Because of its advantages over the previous methods, paging in its various forms is commonly used in many operating systems.

8.5.1 Basic Method

Physical memory is broken into fixed-sized blocks called *frames*. Logical memory is also broken into blocks of the same size called *pages*. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in Figure 8.12. Every address generated by the CPU is divided into two parts: a *page number (p)* and a *page offset (d)*. The page number is used as an index into a *page table*. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical

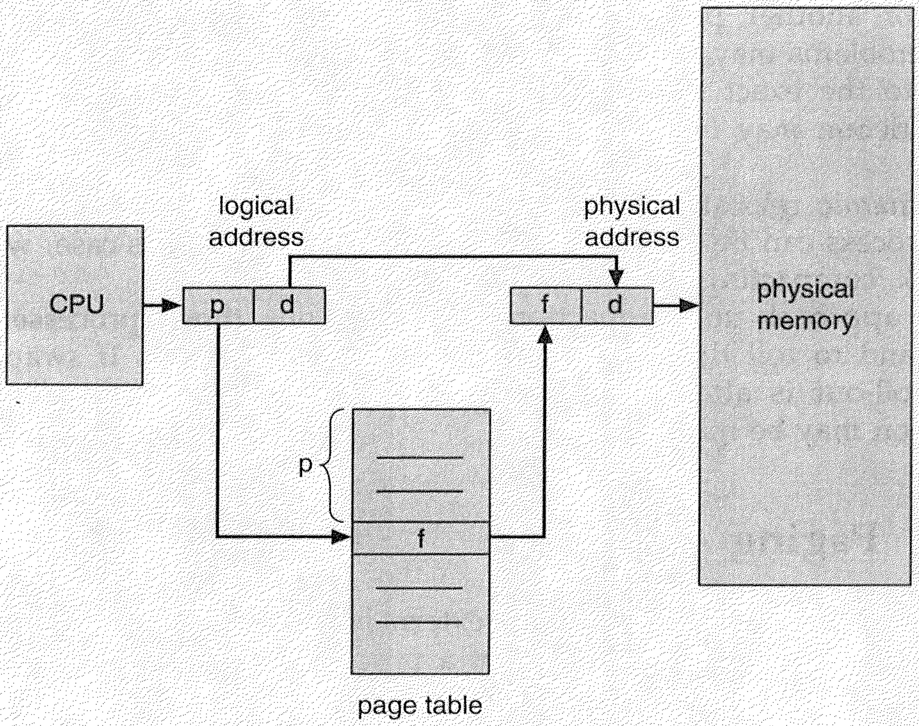


Figure 8.12 Paging hardware.

memory address that is sent to the memory unit. The paging model of memory is shown in Figure 8.13.

The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2 varying between 512 bytes and 8192 bytes per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of logical address space is 2^m, and a page size is 2^n addressing units (bytes or words), then the high-order $m-n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

page number	page offset
p	d
$m-n$	n

where p is an index into the page table and d is the displacement within the page.

For a concrete, although minuscule, example, consider the memory of Figure 8.14. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show an example of how the user's view of memory

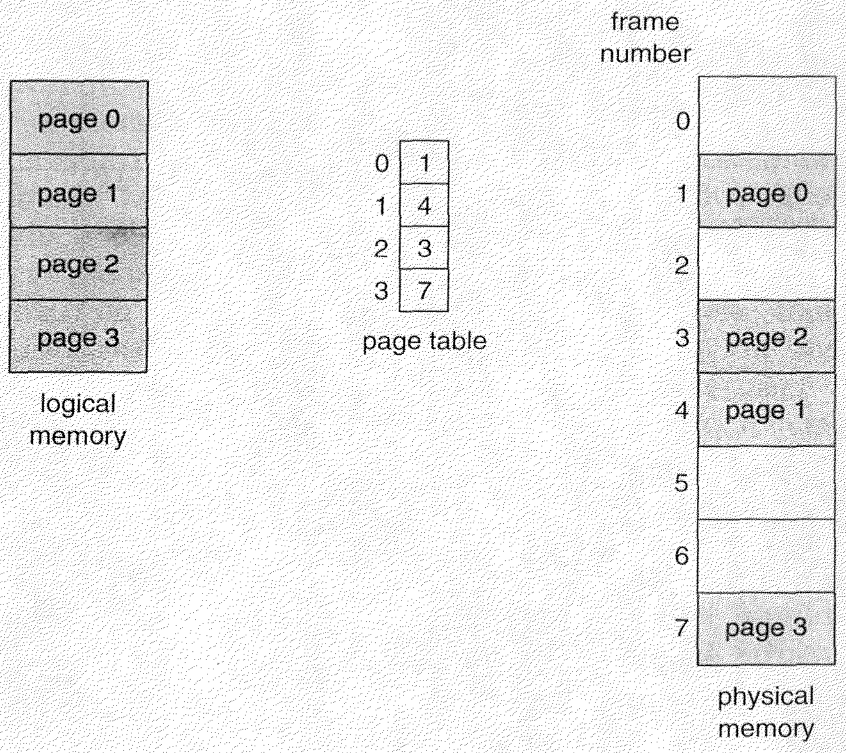


Figure 8.13 Paging model of logical and physical memory.

can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 × 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 × 4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6 × 4) + 0). Logical address 13 maps to physical address 9.

Notice that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. The observant reader will have realized that paging is similar to using a table of base (relocation) registers, one for each frame of memory.

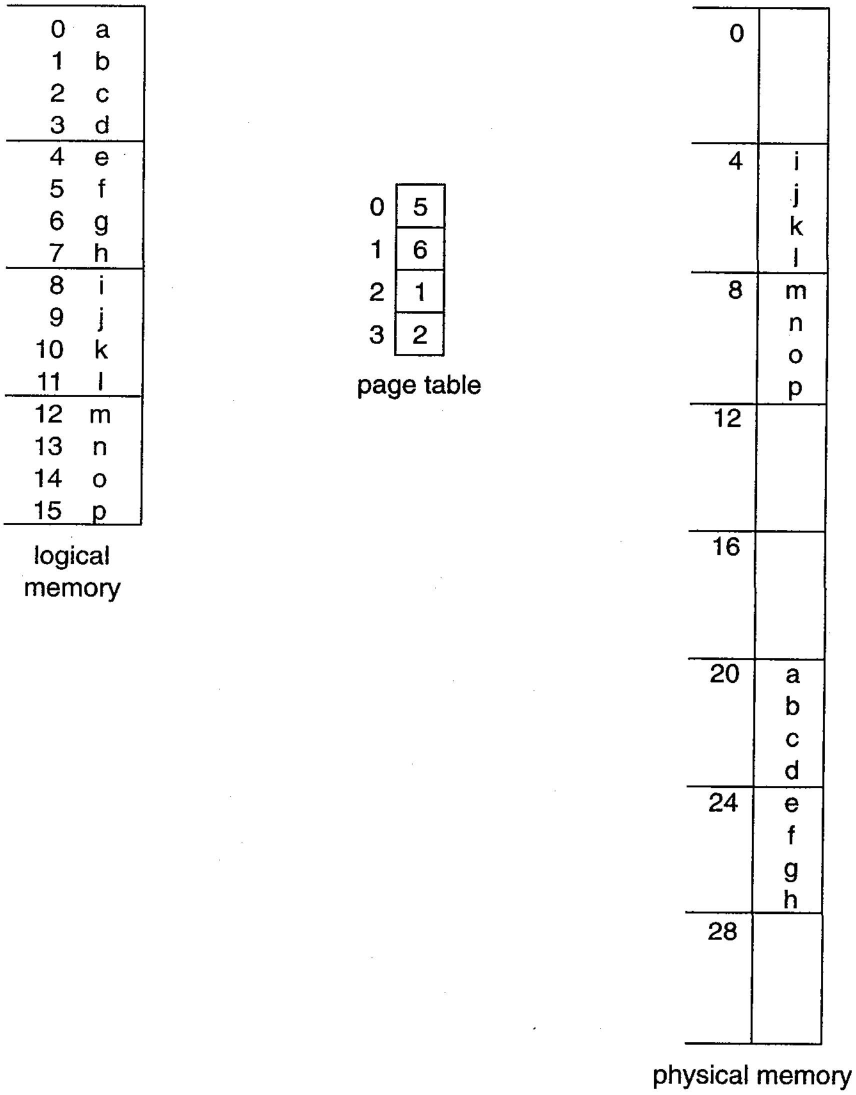


Figure 8.14 Paging example for a 32-byte memory with 4-byte pages.

When we use a paging scheme, we have no external fragmentation: *Any* free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to fall on page boundaries, the *last* frame allocated may not be completely full. For example, if pages are 2048 bytes, a process of 72,766 bytes would need 35 pages plus 1086 bytes. It would be allocated 36 frames, resulting in an internal fragmentation of 2048 − 1086 = 962 bytes. In the worst case, a process would need n pages plus one byte. It would be allocated $n + 1$ frames, resulting in an internal fragmentation of almost an entire frame. If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, there is quite a bit of overhead involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the number of data being transferred is larger (Chapter 12). Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages typically are either 2 or 4 kilobytes.

When a process arrives to be executed, its size, expressed in pages, is examined. Each user page needs one frame. Thus, if the process requires n pages, there must be n frames available in memory. If there are n frames available, they are allocated to this process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table, and so on (Figure 8.15).

An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views that memory as one single contiguous space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

Because the operating system is managing physical memory, it must be aware of the allocation details of physical memory: which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a *frame table*. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

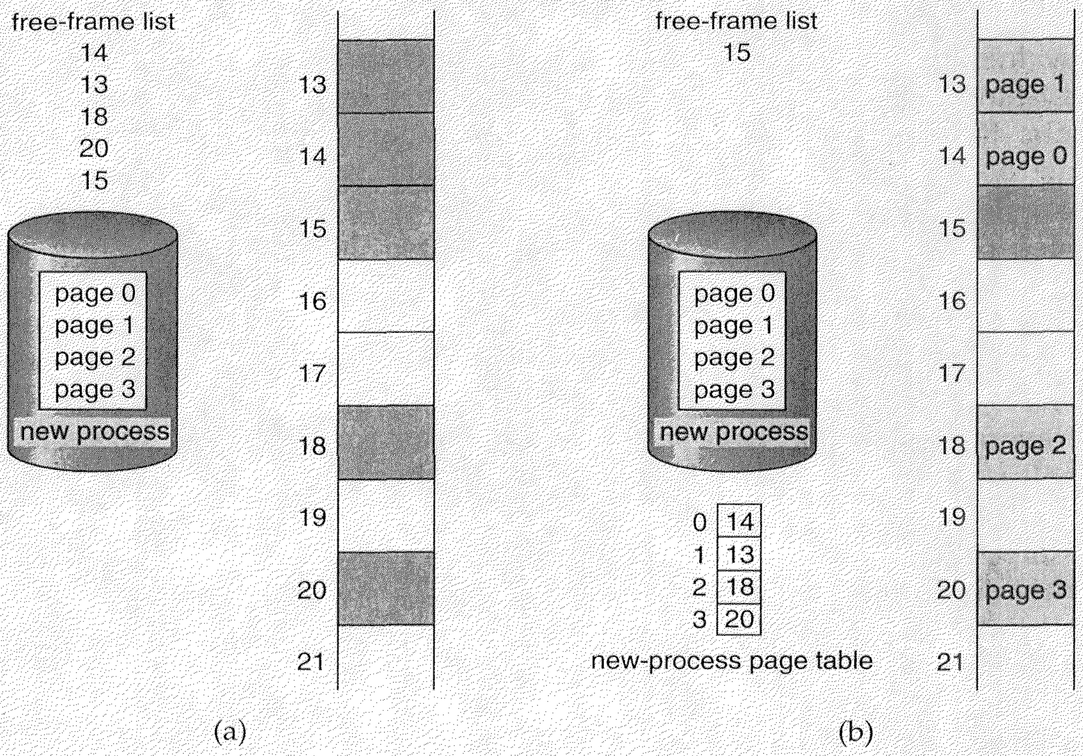


Figure 8.15 Free frames. (a) Before allocation, and (b) after allocation.

In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example) and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

8.5.2 Structure of the Page Table

Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table.

8.5.2.1 Hardware Support

The hardware implementation of the page table can be done in a number of different ways. In the simplest case, the page table is implemented as a set of dedicated *registers*. These registers should be built with very high-speed logic to make the paging address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map. The DEC PDP-11 is an example of such an architecture. The address consists of 16 bits, and the page size is 8K. The page table, thus, consists of eight entries that are kept in fast registers.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a *page-table base register (PTBR)* points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

The problem with this approach is the time required to access a user memory location. If we want to access location *i*, we must first index into the page table, using the value in the PTBR offset by the page number for *i*. This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, *two* memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances. We might as well resort to swapping!

The standard solution to this problem is to use a special, small, fast-lookup hardware cache, variously called *associative registers* or *translation look-aside buffers (TLBs)*. A set of associative registers is built of especially high-speed memory. Each register consists of two parts: a key and a value. When the associative registers are presented with an item, it is compared with all keys simultaneously. If the item is found, the corresponding value field is output. The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB varies between 8 and 2048.

Associative registers are used with page tables in the following way. The associative registers contain only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to a set of associative registers that contain page numbers and their corresponding frame numbers. If the page number is found in the associative registers, its frame number is immediately available and is used to access memory. The whole task may take less than 10 percent longer than it would were an unmapped memory reference used.

If the page number is not in the associative registers, a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (as desired). In addition, we add the page number and frame number to the associative registers, so that they will be found quickly on the next reference (Figure 8.16). If the TLB is already full of entries, the operating system must select one for replacement. Unfortunately, every time a new page table is selected (for instance, each context switch), the TLB must be *flushed* (erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, there could be old entries in the TLB that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that a page number is found in the associative registers is called the *hit ratio.* An 80-percent hit ratio means that we find the desired page number in the associative registers 80 percent of the time. If it takes 20 nanoseconds to search the associative registers, and 100 nanoseconds to access memory, then a mapped memory access takes 120 nanoseconds when the page number is in the associative registers. If we

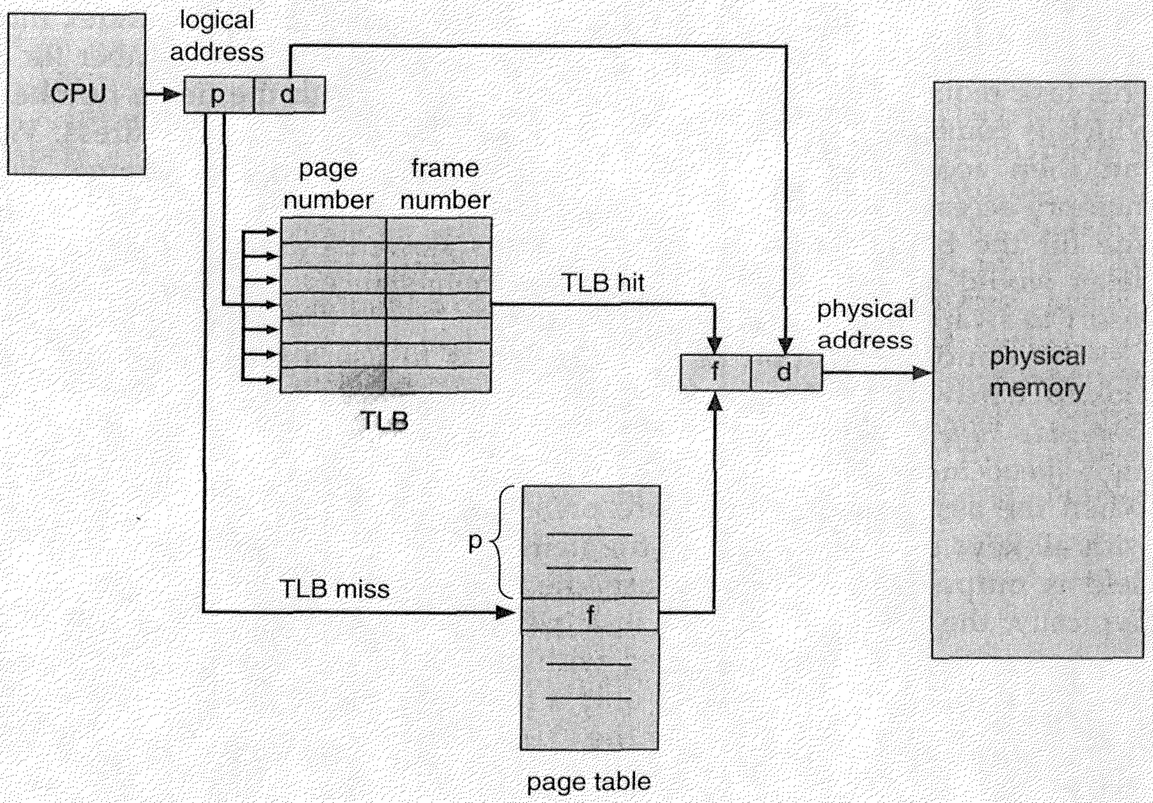


Figure 8.16 Paging hardware with TLB.

fail to find the page number in the associative registers (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds), and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds. To find the *effective memory-access time*, we must weigh each case by its probability:

$$\begin{aligned}\text{effective access time} &= 0.80 \times 120 + 0.20 \times 220\\ &= 140 \text{ nanoseconds.}\end{aligned}$$

In this example, we suffer a 40-percent slowdown in memory access time (from 100 to 140 nanoseconds).

For a 98-percent hit ratio, we have

$$\begin{aligned}\text{effective access time} &= 0.98 \times 120 + 0.02 \times 220\\ &= 122 \text{ nanoseconds.}\end{aligned}$$

This increased hit rate produces only a 22-percent slowdown in memory access time.

The hit ratio is clearly related to the number of associative registers. With the number of associative registers ranging between 16 and 512, a hit ratio of 80 to 98 percent can be obtained. The Motorola 68030 processor (used in Apple Macintosh systems) has a 22-entry TLB. The Intel 80486 CPU (found in some IBM PC compatibles) has 32 registers, and claims a 98-percent hit ratio.

8.5.2.2 Protection

Memory protection in a paged environment is accomplished by protection bits that are associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read and write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (memory-protection violation).

This approach to protection can be expanded easily to provide a finer level of protection. We can create hardware to provide read-only, read-write, or execute-only protection. Or, by providing separate protection bits for each kind of access, any combination of these accesses can be allowed, and illegal attempts will be trapped to the operating system.

One more bit is generally attached to each entry in the page table: a *valid–invalid* bit. When this bit is set to "valid," this value indicates that the associated page is in the process's logical address space, and is thus a legal (valid) page. If the bit is set to "invalid," this value indicates that the page

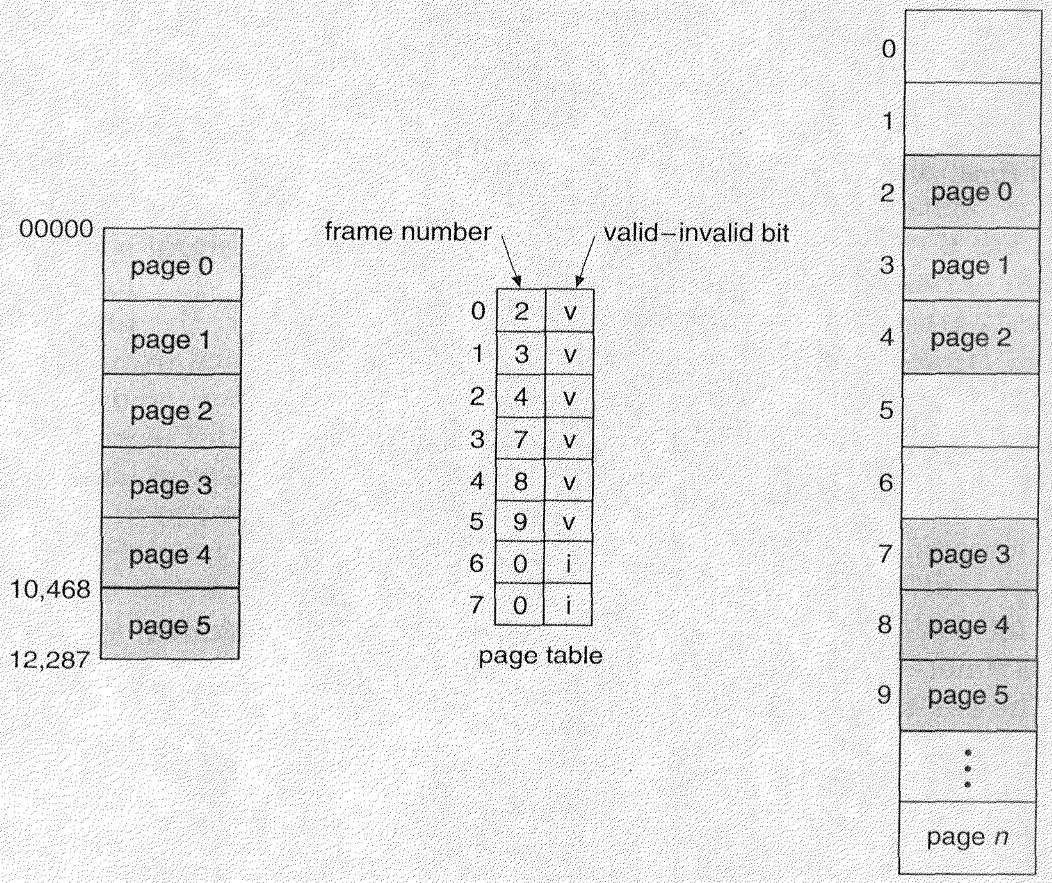


Figure 8.17 Valid (v) or invalid (i) bit in a page table.

is not in the process's logical address space. Illegal addresses are trapped by using the valid–invalid bit. The operating system sets this bit for each page to allow or disallow accesses to that page. For example, in a system with a 14-bit address space (0 to 16,383), we may have a program that should use only addresses 0 to 10,468. Given a page size of 2K, we get the situation shown in Figure 8.17. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, finds that the valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Notice that as the program extends only to address 10,468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12,287 are valid. Only addresses from 12,288 to 16,383 are invalid. This problem is a result of the 2K page size and reflects the internal fragmentation of paging.

Rarely does a process use all of its address range. In fact, many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused, but would take up valuable memory space. Some systems provide hardware, in the form of a *page-table length register (PTLR),* to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

8.5.3 Multilevel Paging

Most modern computer systems support a very large logical address space (2^{32} to 2^{64}). In such an environment the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4K bytes (2^{12}), then a page table may consist of up to 1 million entries (2^{32} / 2^{12}). Because each entry consists of 4 bytes, each process may need up to 4 megabytes of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this is to divide the page table into smaller pieces. There are several different ways to accomplish this.

One way is to use a two-level paging scheme, in which the page table itself is also paged (Figure 8.18). To illustrate this, let us return to our 32-bit machine example, with a page size of 4K bytes. A logical address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page number, and a 10-bit page offset. Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table. The address-translation scheme for this architecture is shown in Figure 8.19. The VAX architecture supports two-level paging. The VAX is a 32-bit machine with page size of 512 bytes. The logical address space of a process is divided into four equal sections, each of which consists of 2^{30} bytes. Each section represents a different part of the logical address space of a process. The first 2 high-order bits of the logical address designate the appropriate section. The next 21 bits represent the logical page number of that section, and the last 9 bits represent an offset in the desired page. By partitioning the page table in this manner, we allow the operating system to leave partitions unused until a process needs them. An address on the VAX architecture is as follows:

section	page	offset
s	p	d
2	21	9

where s designates the section number, p is an index into the page table, and d is the displacement within the page.

A one-level page table for a VAX process using one segment still would be 2^{21} bits * 4 bytes per entry = 8 megabytes. To further reduce main memory use, the VAX pages the user process page tables.

For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. To illustrate this point, suppose that the page size in such a system is 4K bytes (2^{12}). In this case, the page table will consist of up to 2^{52} entries. If we use a two-level paging scheme, then each page in the inner page table would contain 2^9 8-byte entries. (Eight bytes are needed to store the 52-bit pointer to the physical page.) The outer page table will thus consist of $2^{52}/2^9 = 2^{43}$ 8-byte entries, or 2^{46} bytes. Clearly, we would

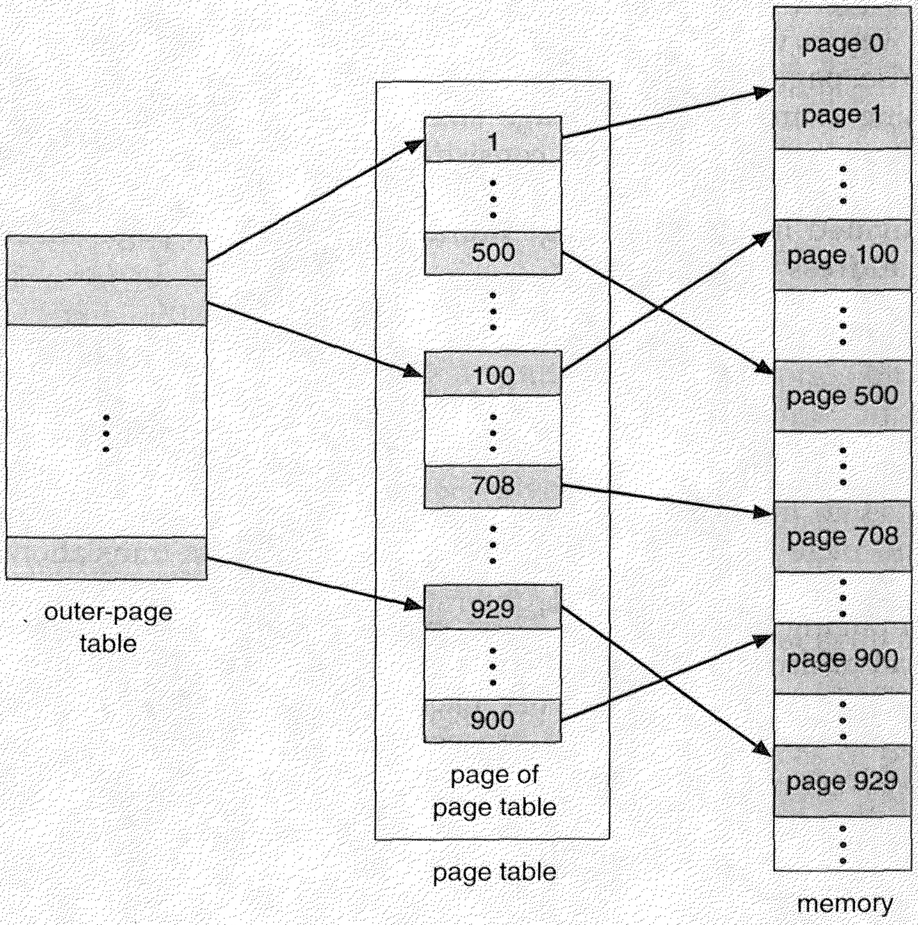


Figure 8.18 A two-level page-table scheme.

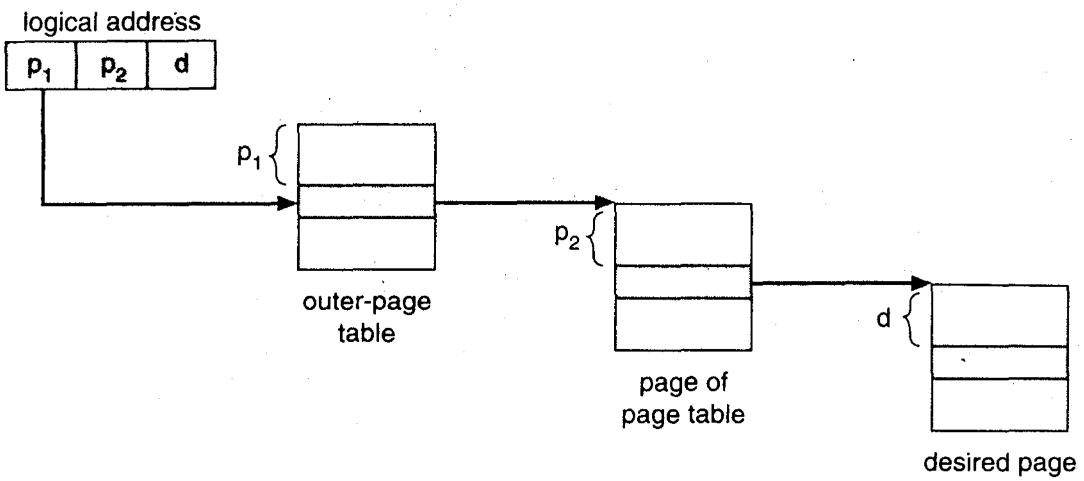


Figure 8.19 Address translation for a two-level 32-bit paging architecture.

not want to allocate the outer page table contiguously in main memory. The obvious solution is to divide the outer page table into smaller pieces. This solution is also used on some 32-bit processors for added flexibility and efficiency.

There are a number of different ways to accomplish this. We can use a three-level paging scheme, where the outer page table itself is also paged (resulting in a second-level outer page). Even with a third level of page tables, each one page in length, a 64-bit address space is daunting, since the second outer page table is still 2^{34} bytes large.

The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged. The SPARC architecture (with 32-bit addressing) supports a three-level paging scheme, whereas the 32-bit Motorola 68030 architecture supports a four-level paging scheme.

How does multilevel paging affect system performance? Given that each level is stored as a separate table in memory, converting a logical address to a physical one may take four memory accesses. We have now quintupled the amount of time needed for one memory access! Caching again pays dividends, however, and performance remains reasonable. Given a cache hit rate of 98 percent, we have

$$\begin{aligned}\text{effective access time} &= 0.98 \times 120 + 0.02 \times 520 \\ &= 128 \text{ nanoseconds.}\end{aligned}$$

Thus, even with the extra levels of table lookup, we have only a 28-percent slowdown in memory access time.

8.5.4 Inverted Page Table

Usually, each process has a page table associated with it. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical-address entry is, and to use that value directly. One of the drawbacks of this scheme is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory, which is required just to keep track of how the other physical memory is being used.

To solve this problem, we can use an *inverted page table*. An inverted page table has one entry for each real page (frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Thus, there is only one page table in the system, and it has only one entry for each page of physical memory. Figure 8.20 shows the operation of an inverted page table. Compare it to Figure 8.12, which depicts a standard page table in operation. Examples of systems using such a scheme are the IBM System/38 computer, the IBM RISC System 6000, IBM RT, and Hewlett-Packard Spectrum workstations.

To illustrate this scheme, we shall describe a simplified version of the implementation of the inverted page table used in the IBM RT. Each virtual address in the system consists of a triple

<process-id, page-number, offset>.

Each inverted page-table entry is a pair <process-id, page-number>. When a memory reference occurs, part of the virtual address, consisting of <process-id, page-number>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found — say, at entry *i* — then the physical address <*i*, offset> is generated. If no match is found, then an illegal address access has been attempted.

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

But do external page tables negate the utility of inverted page tables? Since these tables are referenced only when a page fault is occurring, they do not need to be available quickly. Instead, these tables are themselves paged in and out of memory as necessary. Unfortunately, a page fault

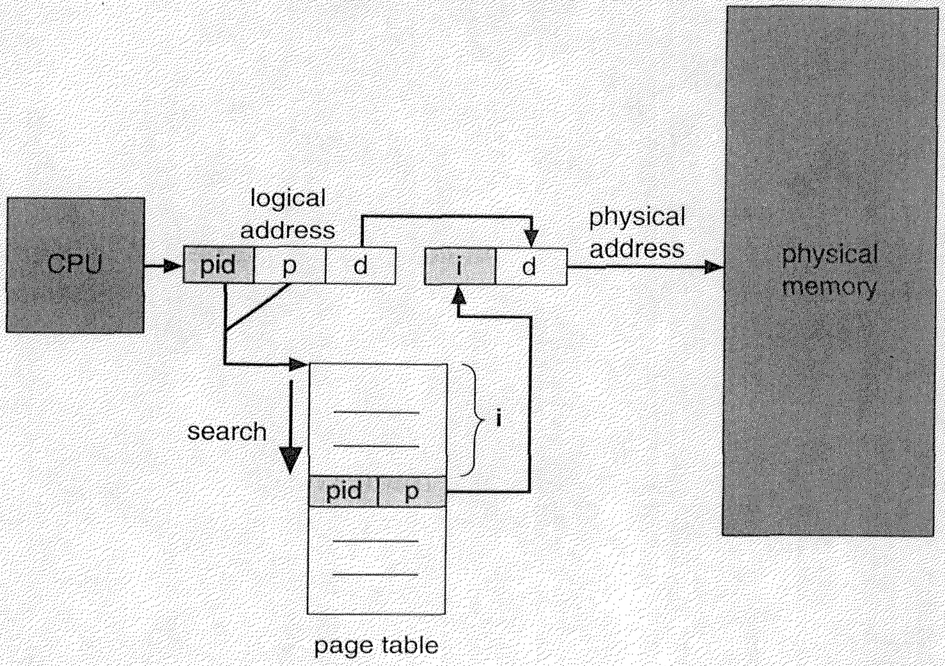


Figure 8.20 Inverted page table.

may now result in the virtual-memory manager causing another page fault as it pages in the external page table it needs to locate the virtual page on the backing store. This special case requires careful handling in the kernel and a delay in the page-lookup processing.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by a physical address, but lookups occur on virtual addresses, the whole table might need to be searched for a match. This search would take far too long. To alleviate this problem, we use a hash table to limit the search to one — or at most a few — page-table entries. Of course, each access to the hash table adds a memory reference to the procedure, so one virtual-memory reference requires at least two real-memory reads: one for the hash-table entry and one for the page table. To improve performance, we use associative memory registers to hold recently located entries. These registers are searched first, before the hash table is consulted.

8.5.5 Shared Pages

Another advantage of paging is the possibility of *sharing* common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150K of code and 50K of data space, we would need 8000K to support the 40 users. If the code is *reentrant*, however, it can be shared, as shown in Figure 8.21. Here we see a three-page editor (each page of size 50K; the large page size is used to simplify the figure) being shared among three processes. Each process has its own data page.

Reentrant code (also called pure code) is non−self-modifying code. If the code is reentrant, then it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process' execution. The data for two different processes will, of course, vary for each process.

Only one copy of the editor needs to be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150K), plus 40 copies of the 50K of data space per user. The total space required is now 2150K, instead of 8000K — a significant savings.

Other heavily used programs also can be shared: compilers, window systems, database systems, and so on. To be sharable, the code must be

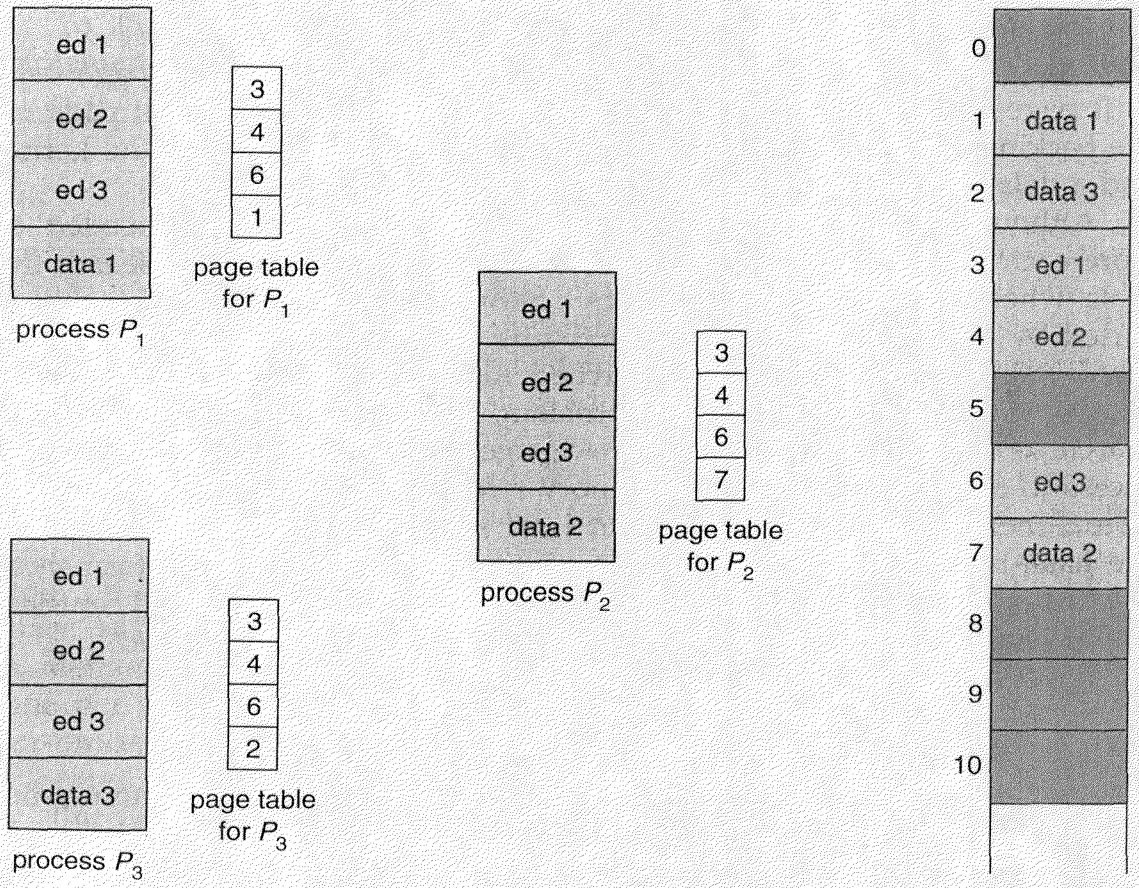


Figure 8.21 Sharing of code in a paging environment.

reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property. This sharing of memory between processes on a system is similar to the way threads share the address space of a task, as described in Chapter 4.

Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is usually implemented as two virtual addresses that are mapped to one physical address. This standard method cannot be used, however, as there is only one virtual page entry for every physical page, so one physical page cannot have the two (or more) shared virtual addresses.

8.6 ■ Segmentation

An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory and the actual physical memory. The user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. The mapping allows differentiation between logical memory and physical memory.

8.6.1 Basic Method

What is the user's view of memory? Does the user think of memory as a linear array of bytes, some containing instructions and others containing data, or is there some other preferred memory view? There is general agreement that the user or programmer of a system does not think of memory as a linear array of bytes. Rather, the user prefers to view memory as a collection of variable-sized segments, with no necessary ordering among segments (Figure 8.22).

Consider how you think of a program when you are writing it. You think of it as a main program with a set of subroutines, procedures, functions, or modules. There may also be various data structures: tables, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. You talk about "the symbol table," "function *Sqrt*," "the main program," without caring what addresses in memory these elements occupy. You are not concerned with whether the symbol table is stored before or after the *Sqrt* function. Each of these segments is of variable length; the length is intrinsically defined by the purpose of the segment in the program. Elements within a segment are identified by their offset from the beginning of the segment: the first statement of the program, the seventeenth entry in the symbol table, the fifth instruction of the *Sqrt* function, and so on.

Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each

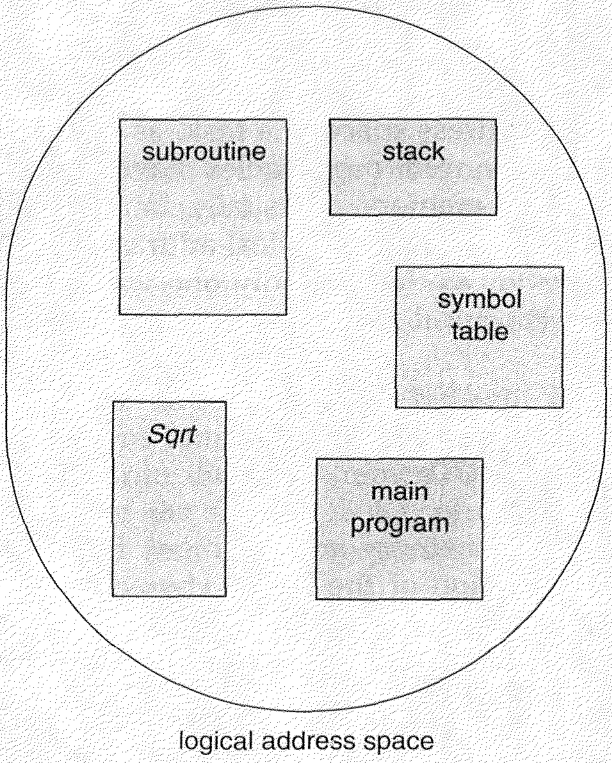


Figure 8.22 User's view of a program.

segment has a name and a length. Addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with paging, where the user specified only a single address, which was partitioned by the hardware into a page number and an offset, all invisible to the programmer.)

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a two tuple:

<segment-number, offset>.

Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program. A Pascal compiler might create separate segments for (1) the global variables; (2) the procedure call stack, to store parameters and return addresses; (3) the code portion of each procedure or function; and (4) the local variables of each procedure and function. A FORTRAN compiler might create a separate segment for each common block. Arrays might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

8.6.2 Hardware

Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. This mapping is effected by a *segment table*. Each entry of the segment table has a segment *base* and a segment *limit*. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

The use of a segment table is illustrated in Figure 8.23. A logical address consists of two parts: a segment number, *s*, and an offset into that segment, *d*. The segment number is used as an index into the segment table. The offset *d* of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). If this offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base–limit register pairs.

As an example, consider the situation shown in Figure 8.24. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical

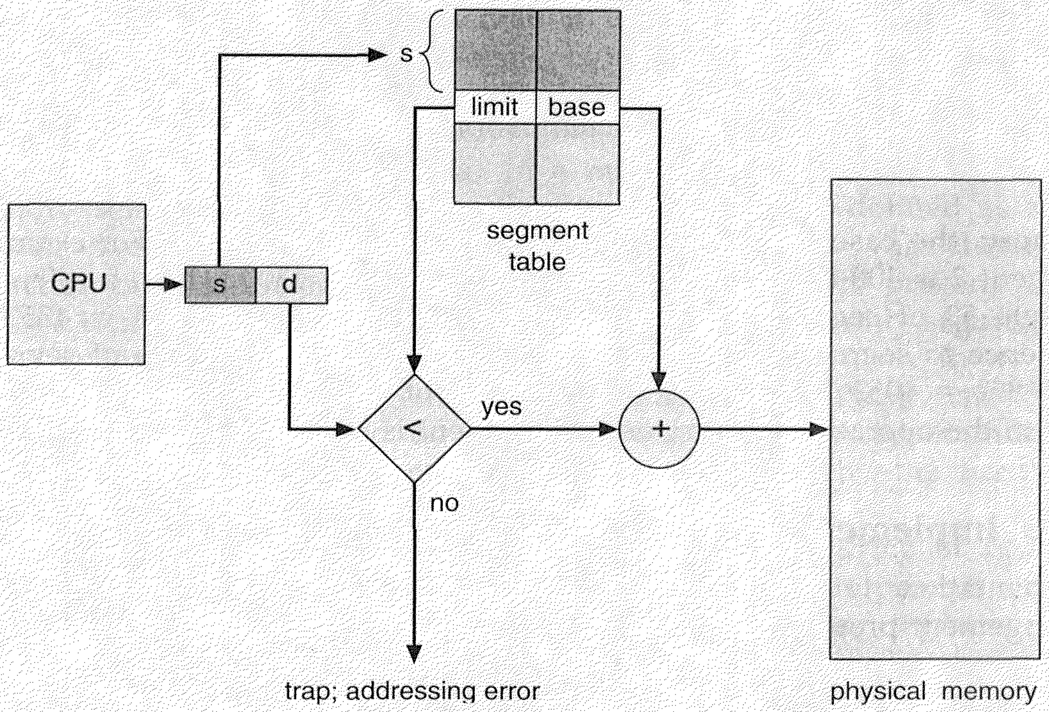
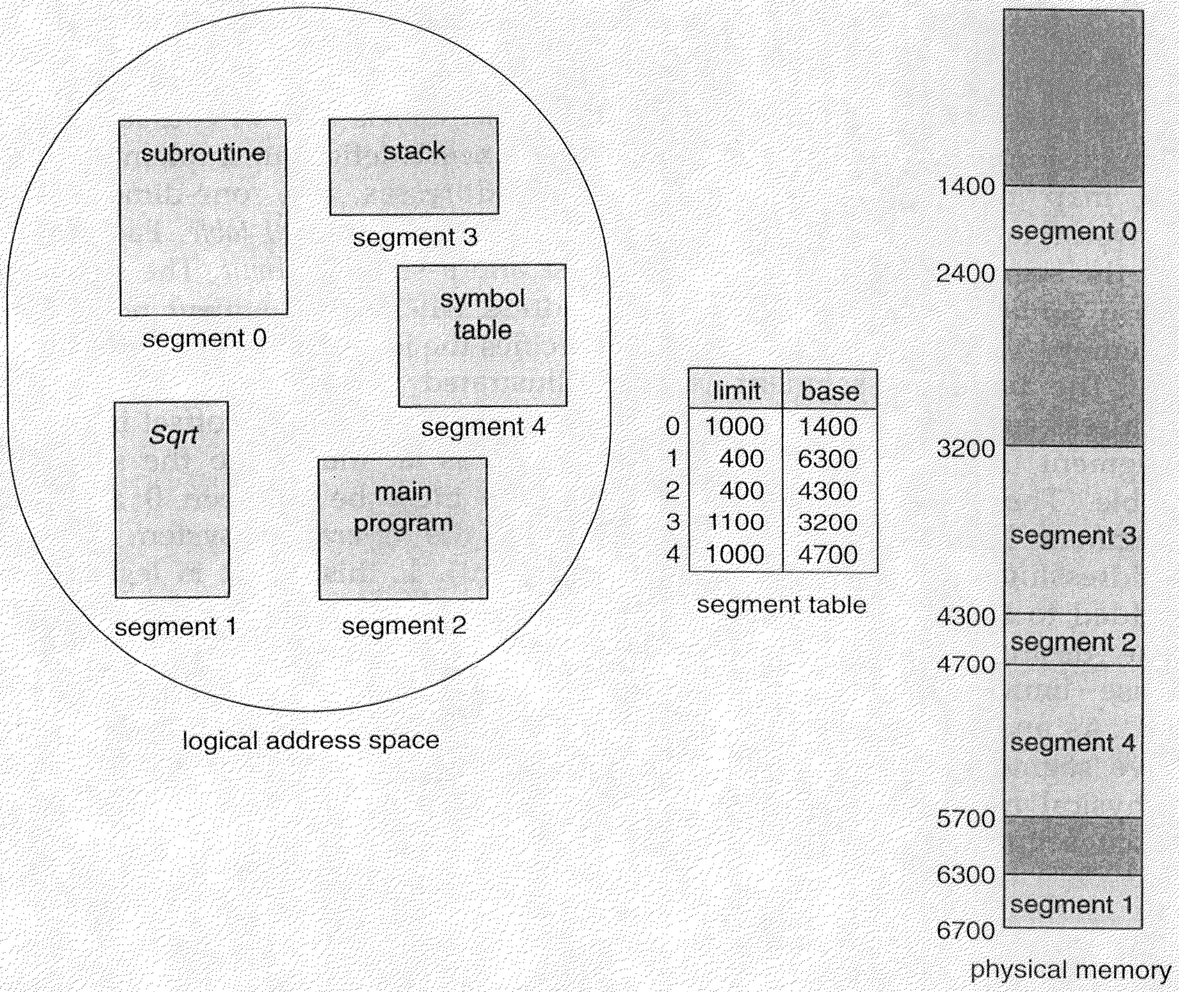


Figure 8.23 Segmentation hardware.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Figure 8.24 Example of segmentation.

memory (the base) and the length of that segment (the limit). For example, segment 2 is 400 bytes long, and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1000 bytes long.

8.6.3 Implementation of Segment Tables

Segmentation is closely related to the partition models of memory management presented earlier, the main difference being that one program may consist of several segments. Segmentation is a more complex concept, however, which is why we are describing it after discussing paging. Like the page table, the segment table can be put either in fast registers or in memory. A segment table kept in registers can be referenced quickly; the addition to the base and comparison with the limit can be done simultaneously to save time.

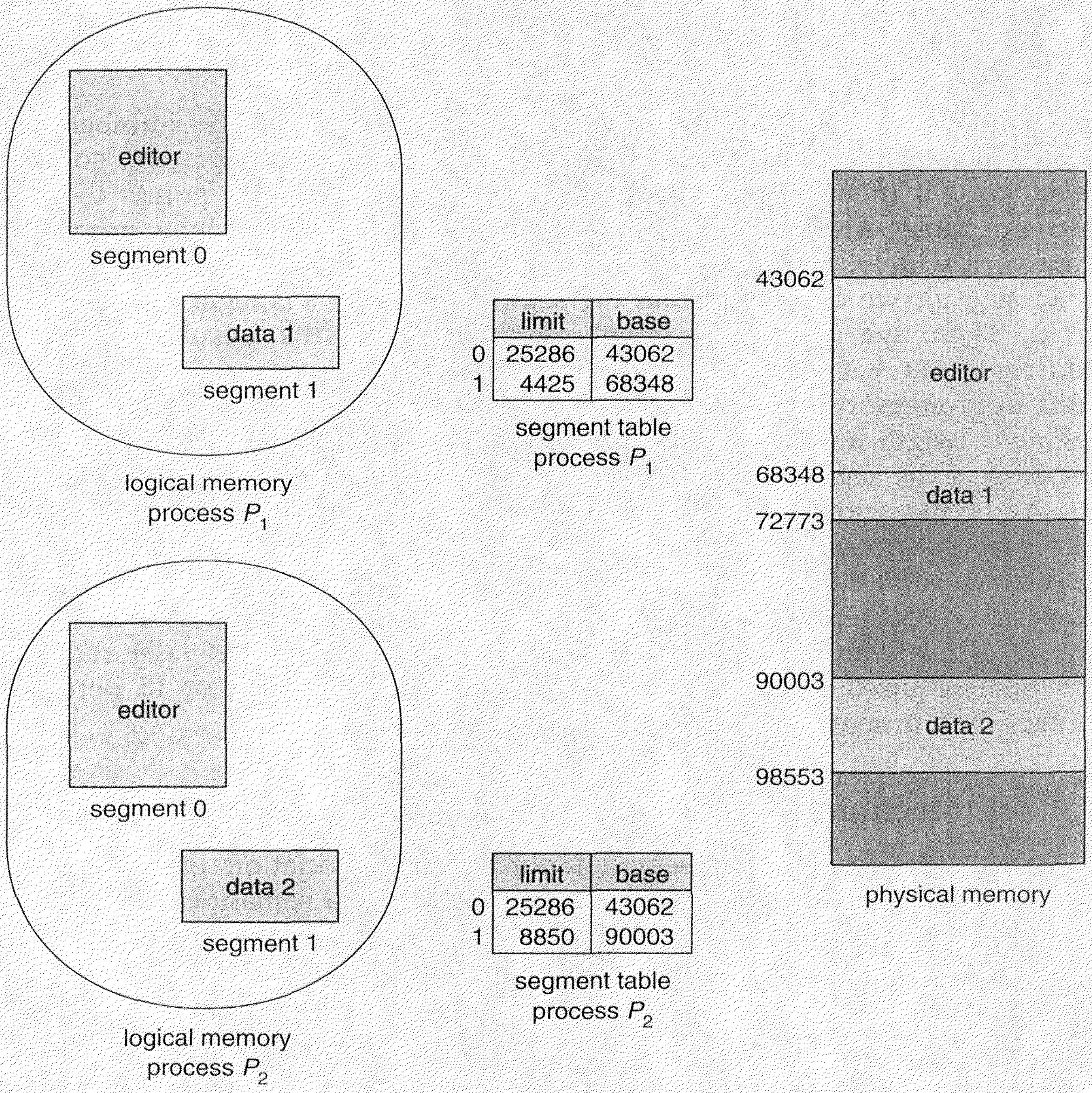
In the case where a program may consist of a large number of segments, it is not feasible to keep the segment table in registers, so we must keep it in memory. A *segment-table base register (STBR)* points to the segment table. Also, because the number of segments used by a program may vary widely, a *segment-table length register (STLR)* is used. For a logical address *(s,d)*, we first check that the segment number *s* is legal (that is, $s <$ STLR). Then, we add the segment number to the STBR, resulting in the address (STBR + *s*) in memory of the segment-table entry. This entry is read from memory and we proceed as before: Check the offset against the segment length and compute the physical address of the desired byte as the sum of the segment base and offset.

As occurs with paging, this mapping requires two memory references per logical address, effectively slowing the computer system by a factor of 2, unless something is done. The normal solution is to use a set of associative registers to hold the most recently used segment-table entries. Again, a relatively small set of associative registers can generally reduce the time required for memory accesses to no more than 10 or 15 percent slower than unmapped memory accesses.

8.6.4 Protection and Sharing

A particular advantage of segmentation is the association of protection with the segments. Because the segments represent a semantically defined portion of the program, it is likely that all entries in the segment will be used the same way. Hence, we have some segments that are instructions, whereas other segments are data. In a modern architecture, instructions are non–self-modifying, so instruction segments can be defined as read-only or execute-only. The memory-mapping hardware will check the protection bits associated with each segment-table entry to prevent illegal accesses to memory, such as attempts to write into a read-only segment, or to use an execute-only segment as data. By placing an array in its own segment, the memory-management hardware will automatically check that array indexes are legal and do not stray outside the array boundaries. Thus, many common program errors will be detected by the hardware before they can cause serious damage.

Another advantage of segmentation involves the *sharing* of code or data. Each process has a segment table associated with its process control block, which the dispatcher uses to define the hardware segment table when this process is given the CPU. Segments are shared when entries in the segment tables of two different processes point to the same physical locations. (Figure 8.25).



	limit	base
0	25286	43062
1	4425	68348

segment table
process P_1

	limit	base
0	25286	43062
1	8850	90003

segment table
process P_2

Figure 8.25 Sharing of segments in a segmented memory system.

The sharing occurs at the segment level. Thus, any information can be shared if it is defined to be a segment. Several segments can be shared, so a program composed of several segments can be shared.

For example, consider the use of a text editor in a time-sharing system. A complete editor might be quite large, composed of many segments. These segments can be shared among all users, limiting the physical memory needed to support editing tasks. Rather than *n* copies of the editor, we need only one copy. For each user, we still need separate, unique segments to store local variables. These segments, of course, would not be shared.

It is also possible to share only parts of programs. For example, common subroutine packages can be shared among many users if they are defined as sharable, read-only segments. Two FORTRAN programs, for instance, may use the same *Sqrt* subroutine, but only one physical copy of the *Sqrt* routine would be needed.

Although this sharing appears simple, there are subtle considerations. Code segments typically contain references to themselves. For example, a conditional jump normally has a transfer address. The transfer address is a segment number and offset. The segment number of the transfer address will be the segment number of the code segment. If we try to share this segment, all sharing processes must define the shared code segment to have the same segment number.

For instance, if we want to share the *Sqrt* routine, and one process wants to make it segment 4 and another wants to make it segment 17, how should the *Sqrt* routine refer to itself? Because there is only one physical copy of *Sqrt,* it must refer to itself in the same way for both users — it must have a unique segment number. As the number of users sharing the segment increases, so does the difficulty of finding an acceptable segment number.

Read-only data segments that contain no physical pointers may be shared as different segment numbers, as may code segments that refer to themselves not directly, but rather only indirectly. For example, conditional branches that specify the branch address as an offset from the current program counter or relative to a register containing the current segment number would allow code to avoid direct reference to the current segment number.

8.6.5 Fragmentation

The long-term scheduler must find and allocate memory for all the segments of a user program. This situation is similar to paging *except* that the segments are of *variable* length; pages are all the same size. Thus, as with the variable-sized partition scheme, memory allocation is a dynamic storage-allocation problem, usually solved with a best-fit or first-fit algorithm.

Segmentation may then cause external fragmentation, when all blocks of free memory are too small to accommodate a segment. In this case, the process may simply have to wait until more memory (or at least a larger hole) becomes available, or compaction may be used to create a larger hole. Because segmentation is by its nature a dynamic relocation algorithm, we can compact memory whenever we want. If the CPU scheduler must wait for one process, due to a memory-allocation problem, it may (or may not) skip through the CPU queue looking for a smaller, lower-priority process to run.

How serious a problem is external fragmentation for a segmentation scheme? Would long-term scheduling with compaction help? The answers to these questions depend mainly on the average segment size. At one extreme, we could define each process to be one segment. This approach reduces to the variable-sized partition scheme. At the other extreme, every byte could be put in its own segment and relocated separately. This arrangement eliminates external fragmentation altogether; however, every byte would need a base register for its relocation, doubling memory use! Of course, the next logical step — fixed-sized, small segments — is paging. Generally, if the average segment size is small, external fragmentation will also be small. (By analogy, consider putting suitcases in the trunk of a car; they never quite seem to fit. However, if you open the suitcases and put the individual items in the trunk, everything fits.) Because the individual segments are smaller than the overall process, they are more likely to fit in the available memory blocks.

8.7 ■ Segmentation with Paging

Both paging and segmentation have their advantages and disadvantages. In fact, of the two most popular microprocessors now being used, the Motorola 68000 line is designed based on a flat address space, whereas the Intel 80X86 family is based on segmentation. Both are merging memory models toward a mixture of paging and segmentation. It is possible to combine these two schemes to improve on each. This combination is best illustrated by two different architectures — the innovative but not widely used MULTICS system and the Intel 386.

8.7.1 MULTICS

In the MULTICS system, a logical address is formed from an 18-bit segment number and a 16-bit offset. Although this scheme creates a 34-bit address space, the segment-table overhead is tolerable; we need only as many segment-table entries as we have segments, as there need not be empty segment-table entries.

However, with segments of 64K words, each of which consists of 36 bits, the average segment size could be large and external fragmentation could be a problem. Even if external fragmentation is not a problem, the search time to allocate a segment, using first-fit or best-fit, could be long. Thus, we may waste memory due to external fragmentation, or waste time due to lengthy searches, or both.

The solution adopted was to *page* the *segments*. Paging eliminates external fragmentation and makes the allocation problem trivial: any empty frame can be used for a desired page. Each page in MULTICS consists of 1K

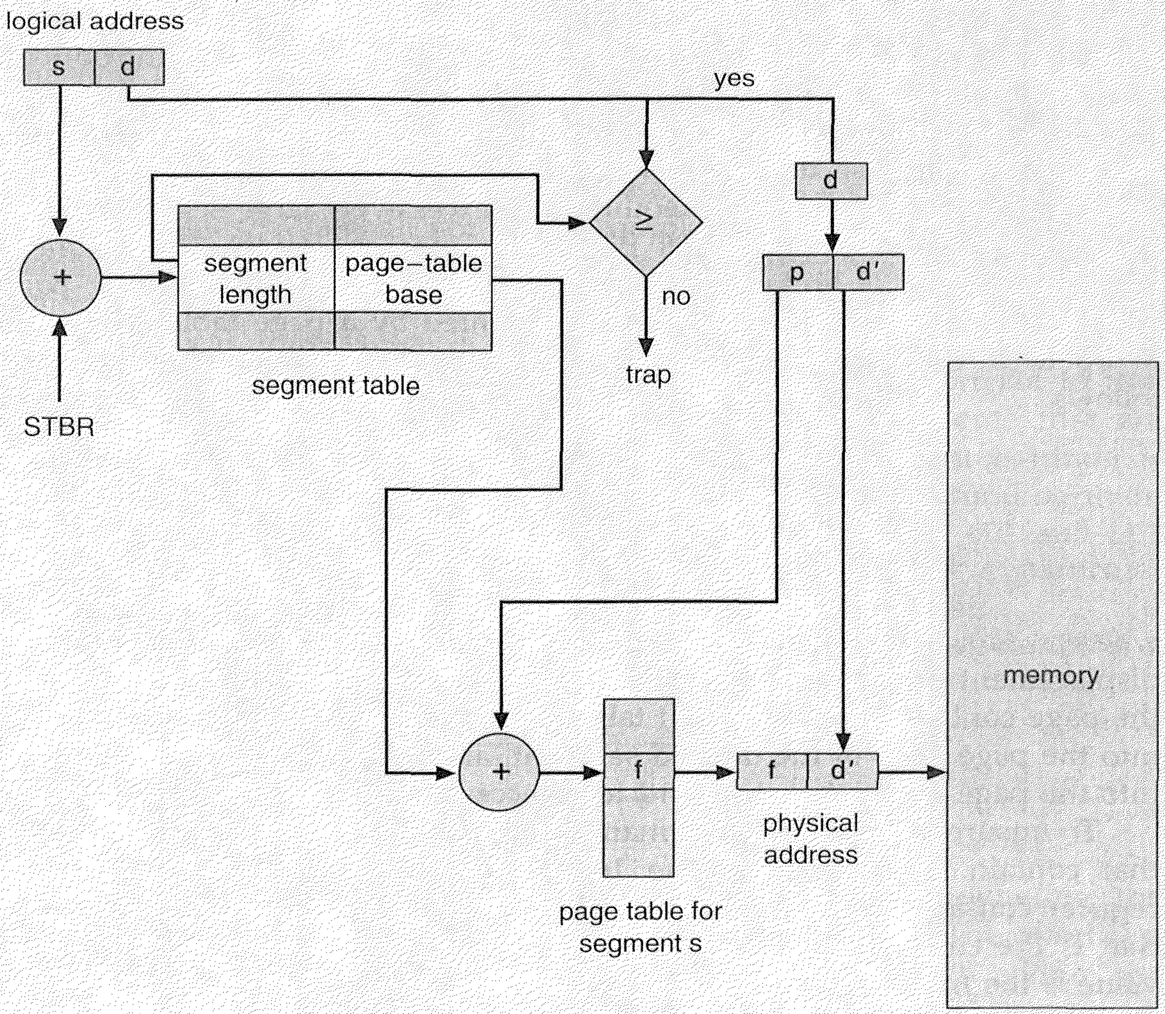


Figure 8.26 Paged segmentation on the GE 645 (MULTICS).

words. Thus, the segment offset (16 bits) is broken into a 6-bit page number and a 10-bit page offset. The page number indexes into the page table to give the frame number. Finally, the frame number is combined with the page offset to form a physical address. The translation scheme is shown in Figure 8.26. Notice that the difference between this solution and pure segmentation is that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.

We must now have a separate page table for each segment. However, because each segment is limited in length by its segment-table entry, the page table does not need to be full sized. It requires only as many entries as are actually needed. As with paging, the last page of each segment generally will not be completely full. Thus, we will have, on the average, one-half page of *internal* fragmentation per segment. Consequently, although we have eliminated external fragmentation, we have introduced internal fragmentation and increased table-space overhead.

In truth, even the paged-segmentation view of MULTICS just presented is simplistic. Because the segment number is an 18-bit quantity, we could have up to 262,144 segments, requiring an excessively large segment table. To ease this problem, MULTICS pages the segment table! The segment number (18 bits) is broken into an 8-bit page number and a 10-bit page offset. Hence, the segment table is represented by a page table consisting of up to 2^8 entries. Thus, in general, a logical address in MULTICS is as follows:

segment number		offset	
s_1	s_2	d_1	d_2
8	10	6	10

where s_1 is an index into the page table of the segment table and s_2 is the displacement within the page of the segment table. Now we have found the page containing the segment table we want. Then, d_1 is a displacement into the page table of the desired segment, and finally, d_2 is a displacement into the page containing the word to be accessed (see Figure 8.27).

To ensure reasonable performance, 16 associative registers are available that contain the address of the 16 most recently referred pages. Each register consists of two parts: a key and a value. The key is a 24-bit field that is the concatenation of a segment number and a page number. The value is the frame number.

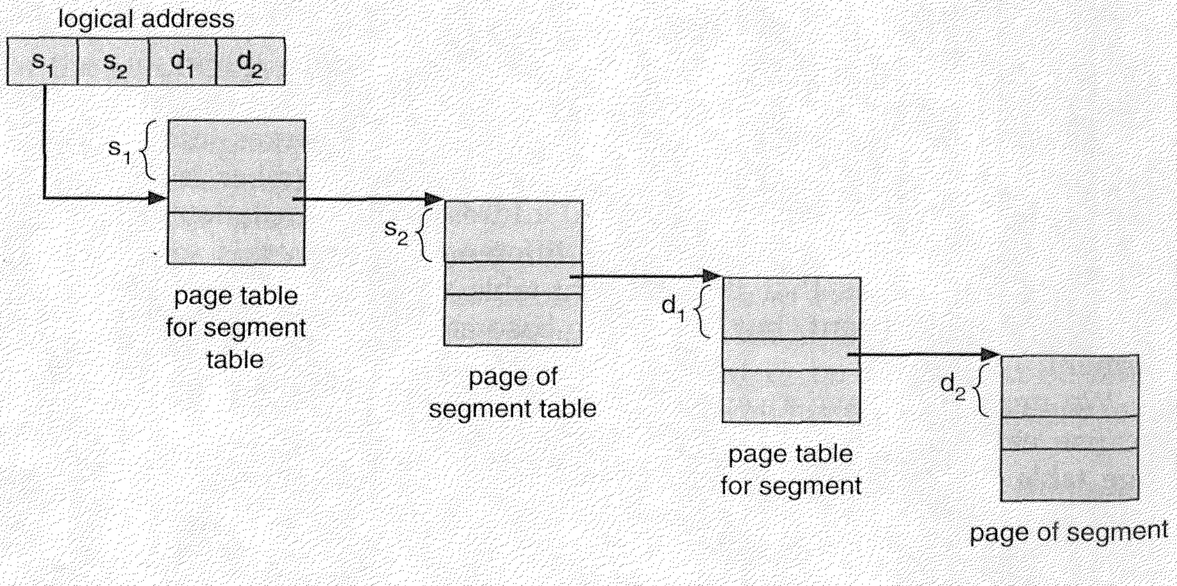


Figure 8.27 Address translation in MULTICS.

8.7.2 OS/2 32-Bit Version

The new IBM OS/2 32-bit version is an operating system running on top of the Intel 386 (and 486) architecture. The 386 uses segmentation with paging for memory management. The maximum number of segments per process is 16K, and each segment can be as large as 4 gigabytes. The page size is 4K bytes. We shall not give a complete description of the memory-management structure of the 386 in this text. Rather, we shall present the major ideas.

The logical address space of a process is divided into two partitions. The first partition consists of up to 8K segments that are private to that process. The second partition consists of up to 8K segments that are shared among all the processes. Information about the first partition is kept in the *local descriptor table (LDT),* information about the second partition is kept in the *global descriptor table (GDT).* Each entry in the LDT and GDT table consists of 8 bytes, with detailed information about a particular segment including the base location and length of that segment.

The logical address is a pair (selector, offset), where the selector is a 16-bit number:

s	g	p
13	1	2

in which s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection. The offset is a 32-bit number specifying the location of the byte (word) within the segment in question.

The machine has six segment registers, allowing six segments to be addressed at any one time by a process. It has six 8-byte microprogram registers to hold the corresponding descriptors from either the LDT or GDT. This cache lets the 386 avoid having to read the descriptor from memory for every memory reference.

The physical address on the 386 is 32 bits long and is formed as follows. The select register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment in question are used to generate a *linear address.* First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address.

As pointed out previously, each segment is paged, and each page is 4K bytes. A page table may thus consist of up to 1 million entries. Because each entry consists of 4 bytes, each process may need up to 4 megabytes of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. The solution adopted in the 386 is to use a two-level paging scheme. The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit page table pointer. The logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

The address-translation scheme for this architecture is similar to the scheme shown in Figure 8.19. The Intel address translation is shown in more detail in Figure 8.28. So that the efficiency of physical-memory use can be improved, Intel 386 page tables can be swapped to disk. In this case, an invalid bit is used in the page-directory entry to indicate whether the table to which the entry is pointing is in memory or on disk. If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.

8.8 ■ Summary

Memory-management algorithms for multiprogrammed operating systems range from the simple single-user system approach to paged segmentation. The greatest determinant of the method used in a particular system is the hardware provided. Every memory address generated by the CPU must be checked for legality and possibly mapped to a physical address. The checking cannot be implemented (efficiently) in software. Hence, we are constrained by the hardware available.

The memory-management algorithms discussed (contiguous allocation, paging, segmentation, and combinations of paging and segmentation) differ in many aspects. The following list indicates some important considerations that you should use in comparing different memory-management strategies:

- **Hardware support:** A simple base register or a pair of base and limit registers is sufficient for the single and multiple partition schemes, whereas paging and segmentation need mapping tables to define the address map.

- **Performance:** As the algorithm becomes more complex, the time required to map a logical address to a physical address increases. For the simple systems, we need only to compare or add to the logical address — operations that are fast. Paging and segmentation can be as

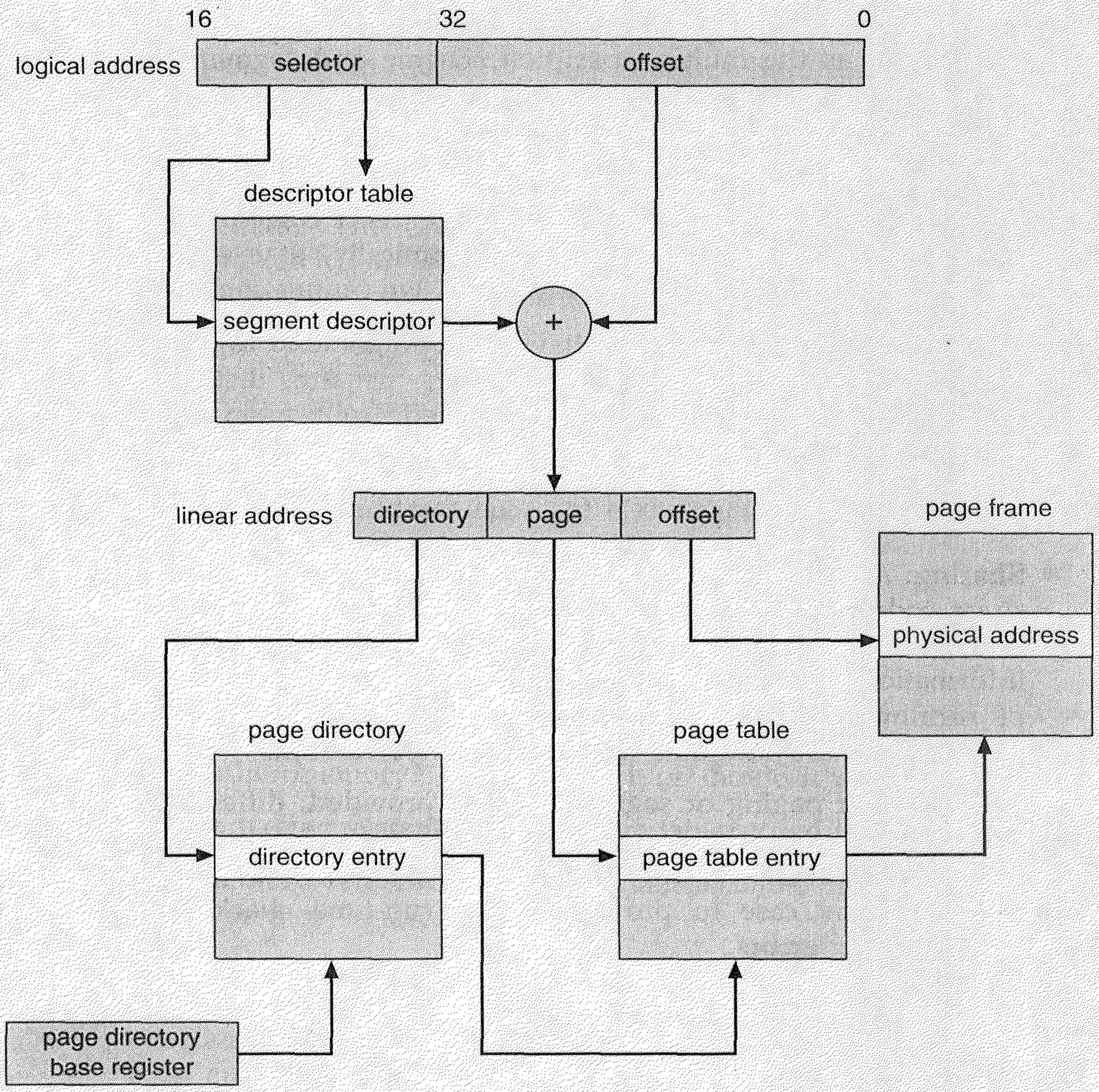


Figure 8.28 Intel 80386 address translation.

fast if the table is implemented in fast registers. If the table is in memory, however, user memory accesses can be degraded substantially. A set of associative registers can reduce the performance degradation to an acceptable level.

- **Fragmentation:** A multiprogrammed system will generally perform more efficiently with a higher level of multiprogramming. For a given set of processes, we can increase the multiprogramming level only by packing more processes into memory. To accomplish this task, we must reduce memory waste or fragmentation. Systems with fixed-sized allocation units, such as the single-partition scheme and paging, suffer from internal fragmentation. Systems with variable-sized allocation units, such as the multiple-partition scheme and segmentation, suffer from external fragmentation.

- **Relocation:** One solution to the external-fragmentation problem is compaction. Compaction involves shifting a program in memory without the program noticing the change. This consideration requires that logical addresses be relocated dynamically, at execution time. If addresses are relocated only at load time, we cannot compact storage.
- **Swapping:** Any algorithm can have swapping added to it. At intervals determined by the operating system, usually dictated by CPU-scheduling policies, processes are copied from main memory to a backing store, and later are copied back to main memory. This scheme allows more processes to be run than can be fit into memory at one time.
- **Sharing:** Another means of increasing the multiprogramming level is to share code and data among different users. Sharing generally requires that either paging or segmentation be used, to provide small packets of information (pages or segments) that can be shared. Sharing is a means of running many processes with a limited amount of memory, but shared programs and data must be designed carefully.
- **Protection:** If paging or segmentation is provided, different sections of a user program can be declared execute-only, read-only, or read–write. This restriction is necessary with shared code or data, and is generally useful in any case to provide simple run-time checks for common programming errors.

■ Exercises

8.1 Explain the difference between logical and physical addresses.

8.2 Explain the following allocation algorithms:

a. First-fit

b. Best-fit

c. Worst-fit

8.3 When a process is rolled out of memory, it loses its ability to use the CPU (at least for a while). Describe another situation where a process loses its ability to use the CPU, but where the process does not get rolled out.

8.4 Explain the difference between internal and external fragmentation.

8.5 Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?

8.6 Consider a system where a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base–limit register pairs are provided: one for instructions and one for data. The instruction base–limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

8.7 Why are pages sizes always powers of 2?

8.8 Consider a logical address space of eight pages of 1024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in the physical address?

8.9 Why is it that, on a system with paging, a process cannot access memory it does not own? How could the operating system allow access to other memory? Why should it or should it not?

8.10 Consider a paging system with the page table stored in memory.

a. If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?

b. If we add associative registers, and 75 percent of all page-table references are found in the associative registers, what is the effective memory reference time? (Assume that finding a page-table entry in the associative registers takes zero time, if the entry is there.)

8.11 What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What would the effect of updating some byte in the one page be on the other page?

8.12 Why are segmentation and paging sometimes combined into one scheme?

8.13 Describe a mechanism by which one segment could belong to the address space of two different processes.

8.14 Explain why it is easier to share a reentrant module using segmentation than it is to do so when pure paging is used.

8.15 Sharing segments among processes without requiring the same segment number is possible in a dynamically-linked segmentation system.

a. Define a system that allows static linking and sharing of segments without requiring that the segment numbers be the same.

b. Describe a paging scheme that allows pages to be shared without requiring that the page numbers be the same.

8.16 Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0,430

b. 1,10

c. 2,500

d. 3,400

e. 4,112

8.17 Consider the Intel address translation scheme shown in Figure 8.28.

a. Describe all the steps that are taken by the Intel 80386 in translating a logical address into a physical address.

b. What are the advantages to the operating system of hardware that provides such complicated memory translation hardware?

c. Are there any disadvantages to this address translation system?

8.18 In the IBM/370, memory protection is provided through the use of *keys*. A key is a 4-bit quantity. Each 2K block of memory has a key (the storage key) associated with it. The CPU also has a key (the protection key) associated with it. A store operation is allowed only if both keys are equal, or if either is zero. Which of the following memory-management schemes could be used successfully with this hardware?

a. Bare machine

b. Single-user system

c. Multiprogramming with a fixed number of processes

d. Multiprogramming with a variable number of processes

e. Paging

f. Segmentation

Bibliographic Notes

Dynamic storage allocation was discussed by Knuth [1973, Section 2.5], who found through simulation results that first-fit is generally superior to best-fit. Additional discussions were offered by Shore [1975], Bays [1977], Stephenson [1983] and Brent [1989]. An adaptive exact-fit storage-management scheme was presented by Oldehoeft and Allan [1985]. Discussions concerning the 50-percent rule were offered by Knuth [1973].

The concept of paging can be credited to the designers of the Atlas system, which has been described by Kilburn et al. [1961] and Howarth et al. [1961]. The concept of segmentation was first discussed by Dennis [1965]. Paged segmentation was first supported in the GE 645, on which MULTICS was originally implemented [Organick 1972].

Inverted page tables were discussed in an article about the IBM RT storage manager by Chang and Mergen [1988].

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

The Motorola 68000 microprocessor family was described in Motorola [1989a]. The Intel 8086 was described in Intel [1985a]. The Intel 80386 paging hardware was described in Intel [1986]. Tanenbaum [1992] also discussed Intel 80386 paging. The new Intel 80486 hardware was covered in Intel [1989].

CHAPTER 9

VIRTUAL MEMORY

In Chapter 8, we discussed various memory-management strategies that have been used in computer systems. All these strategies have the same goal: to keep many processes in memory simultaneously to allow multiprogramming. However, they tend to require the entire process to be in memory before the process can execute.

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Further, it abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from concern over memory storage limitations. Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly. In this chapter, we discuss virtual memory in the form of demand paging, and examine its complexity and cost.

9.1 ■ Background

The memory-management algorithms of Chapter 8 are necessary because of one basic requirement: The instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Overlays and dynamic loading can help ease this restriction, but they generally require special precautions and extra effort by the programmer. This restriction seems both necessary and reasonable, but it is also unfortunate, since it limits the size of a program to the size of physical memory.

In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance,

- Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements. An assembler symbol table may have room for 3000 symbols, although the average program has less than 200 symbols.
- Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers which balance the budget have not been used in years.

Even in those cases where the entire program is needed, it may not all be needed at the same time (such is the case with overlays, for example).

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

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

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

Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 9.1). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available, or about what code can be placed in overlays, but can concentrate instead on the problem to be programmed. On systems which support virtual memory, overlays have virtually disappeared.

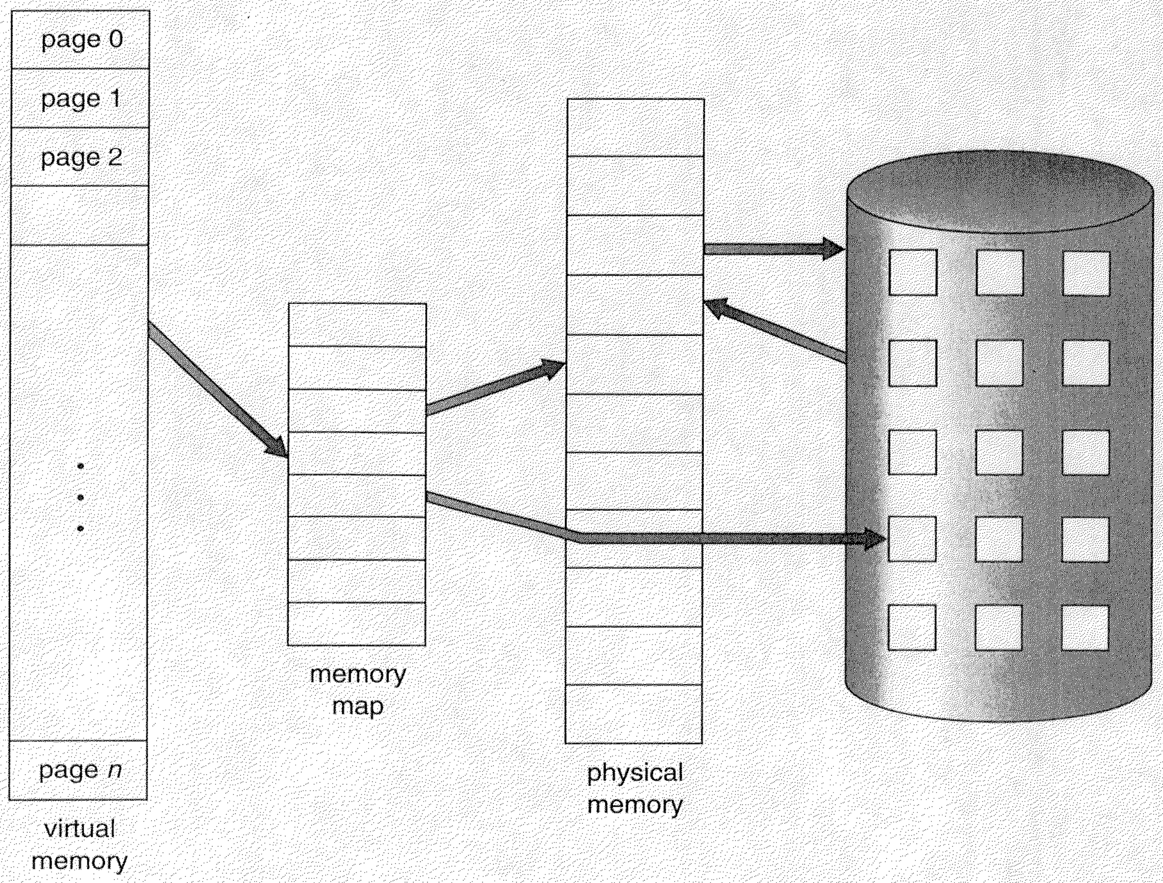


Figure 9.1 Diagram showing virtual memory larger than physical memory.

Virtual memory is commonly implemented by *demand paging*. It can also be implemented in a segmentation system. Several systems provide a paged segmentation scheme, where segments are broken into pages. Thus, the user view is segmentation, but the operating system can implement this view with demand paging. *Demand segmentation* can also be used to provide virtual memory. Burroughs' computer systems have used demand segmentation. The IBM OS/2 operating system also uses demand segmentation. However, segment-replacement algorithms are more complex than are page-replacement algorithms because the segments have variable sizes.

9.2 ■ Demand Paging

A demand-paging system is similar to a paging system with swapping (Figure 9.2). Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a

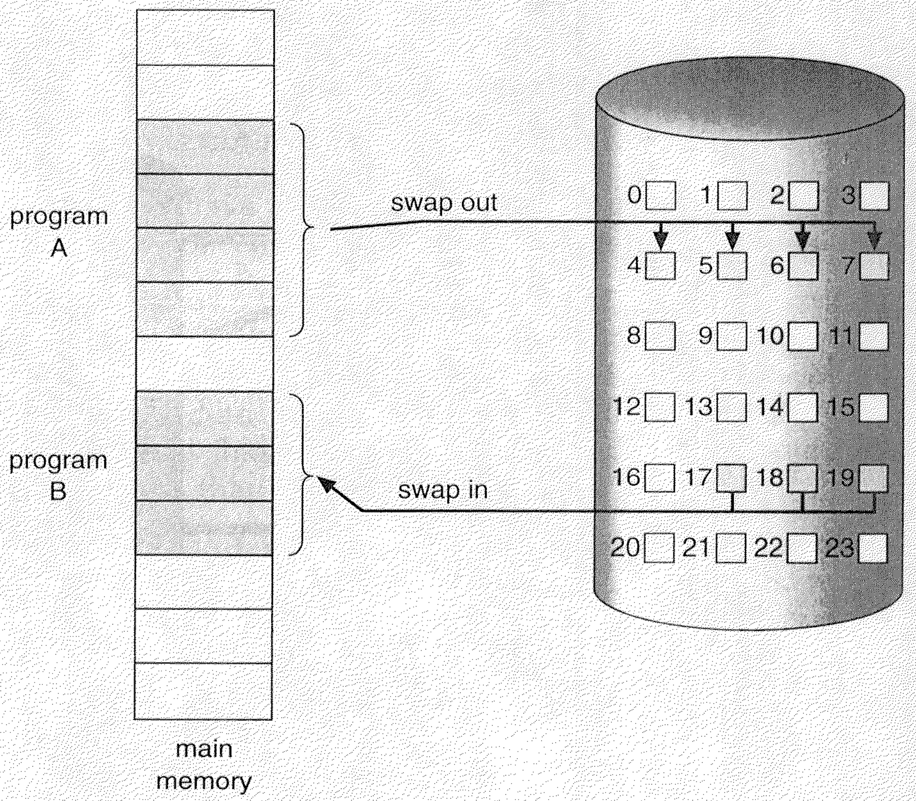


Figure 9.2 Transfer of a paged memory to contiguous disk space.

lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages, rather than one large contiguous address space, the use of the term *swap* is technically incorrect. A swapper manipulates entire processes, whereas a *pager* is concerned with the individual pages of a process. We shall thus use the term *pager*, rather than *swapper*, in connection with demand paging.

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

With this scheme, we need some form of hardware support to distinguish between those pages that are in memory and those pages that are on the disk. The *valid–invalid* bit scheme described in Section 8.5.2 can be used for this purpose. This time, however, when this bit is set to "valid," this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid," this value indicates that the page is either not valid (that is, not in the logical address space of the process), or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid, or contains the address of the page on disk. This situation is depicted in Figure 9.3.

Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we had brought in all pages. While the process executes and accesses pages that are *memory resident*, execution proceeds normally.

But what happens if the process tries to use a page that was not brought into memory? Access to a page marked invalid causes a *page-fault* trap. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory (in an attempt to minimize disk-transfer

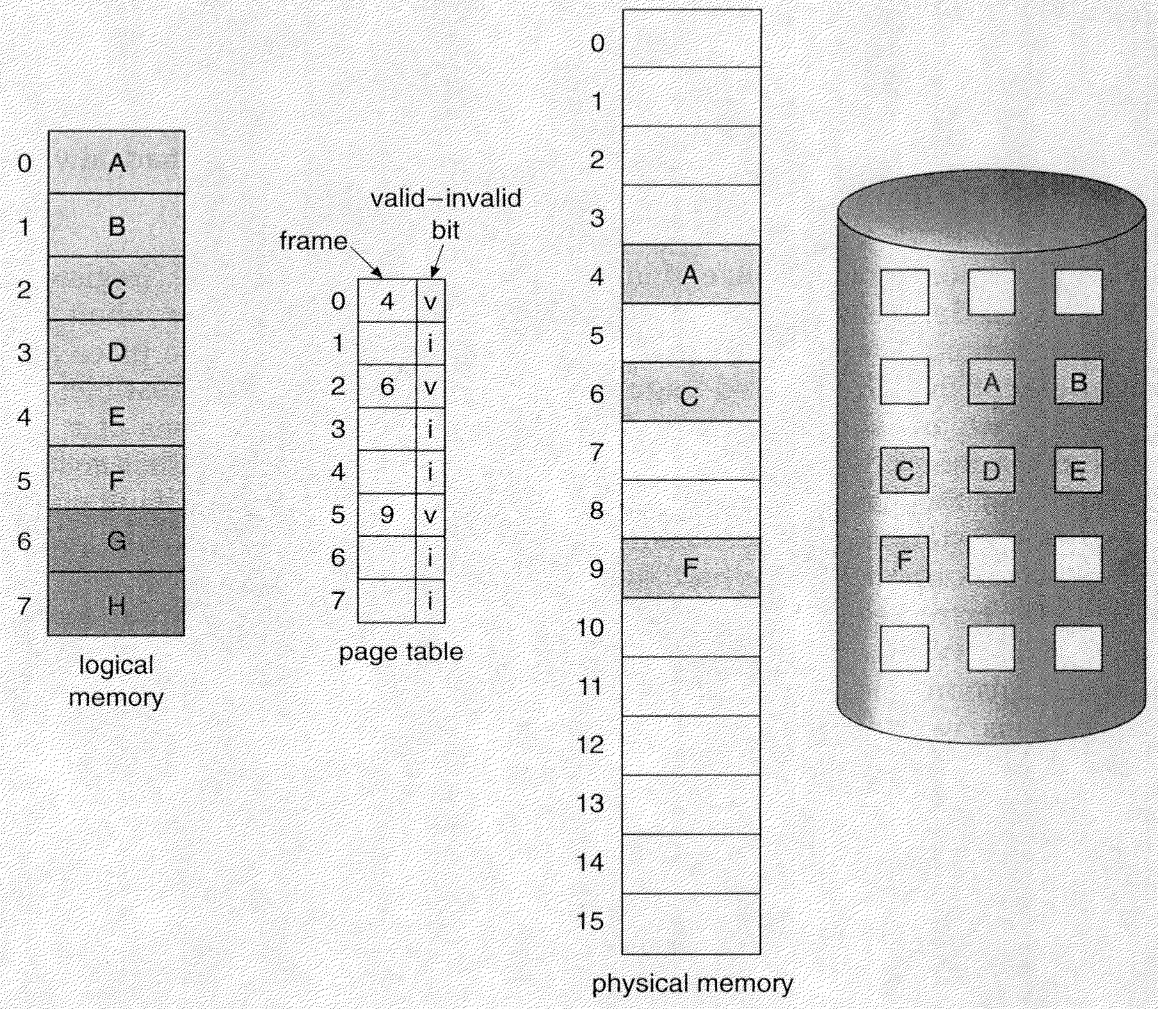


Figure 9.3 Page table when some pages are not in main memory.

overhead and memory requirements), rather than an invalid address error as a result of an attempt to use an illegal memory address (such as an incorrect array subscript). We must therefore correct this oversight. The procedure for handling this page fault is simple (Figure 9.4):

1. We check an internal table (usually kept with the process control block) for this process, to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page in the latter.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.

It is important to realize that, because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we can restart the process in *exactly* the same place and state, except that the desired page is now in memory and is accessible. In this way, we are able to execute a process, even though portions of it are not (yet) in memory. When the process tries to access locations that are not in memory, the hardware traps to the operating system (page fault). The operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

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

Theoretically, some programs may access several new pages of memory with each instruction execution, (one page for the instruction and many for data) possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately,

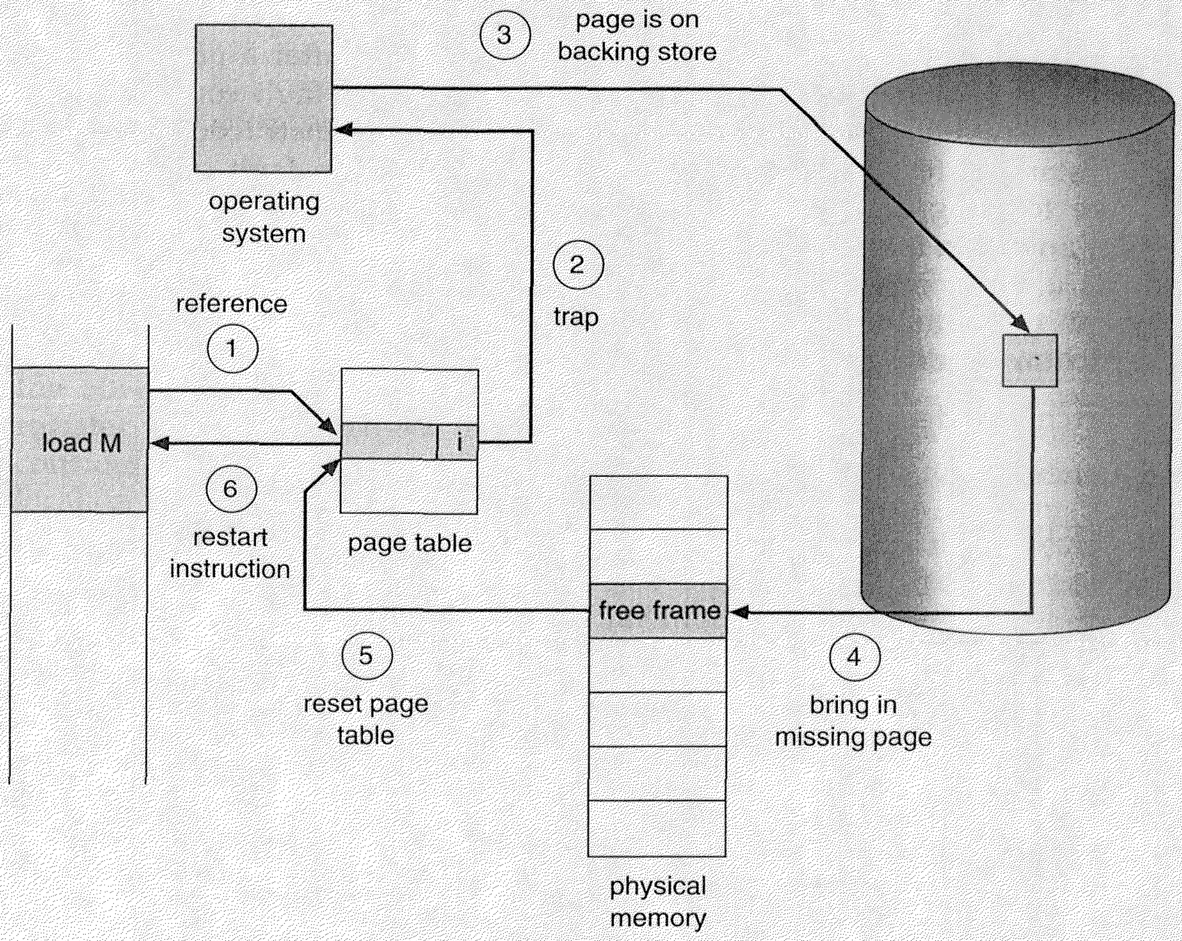


Figure 9.4 Steps in handling a page fault.

analysis of running processes show that this behavior is exceedingly unlikely. Programs tend to have *locality of reference*, described in Section 9.7.1, which results in reasonable performance from demand paging.

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

- **Page table:** This table has the ability to mark an entry invalid through a valid–invalid bit or special value of protection bits.
- **Secondary memory:** This memory holds those pages not in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as *swap space* or *backing store*. Swap space allocation is further discussed in Chapter 12.

In addition to this hardware support, considerable software is needed, as we shall see.

Some additional architectural constraints must be imposed. A crucial issue is the need to be able to restart any instruction after a page fault. In most cases, this requirement is easy to meet. A page fault could occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must re-fetch the instruction, decode it again, and then fetch the operand.

As a worst case, consider a three-address instruction such as ADD the content of A to B placing the result in C. The steps to execute this instruction would be

1. Fetch and decode the instruction (ADD).
2. Fetch A.
3. Fetch B.
4. Add A and B.
5. Store the sum in C.

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

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

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

A similar architectural problem occurs in machines that use special addressing modes, including autodecrement and autoincrement modes (for example, the PDP-11). These addressing modes use a register as a pointer and automatically decrement or increment the register as indicated. Autodecrement automatically decrements the register *before* using its contents as the operand address; autoincrement automatically increments the register *after* using its contents as the operand address. Thus, the instruction

MOV (R2)+,−(R3)

copies the contents of the location pointed to by register 2 into the location pointed to by register 3. Register 2 is incremented (by 2 for a word, since the PDP-11 is a byte-addressable computer) after it is used as a pointer; register 3 is decremented (by 2) before it is used as a pointer. Now consider what will happen if we get a fault when trying to store into the location pointed to by register 3. To restart the instruction, we must reset the two registers to the values they had before we started the execution of the instruction. One solution is to create a new special status register to record the register number and amount modified for any register that is changed during the execution of an instruction. This status register allows the operating system to "undo" the effects of a partially executed instruction that causes a page fault.

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

9.3 ■ Performance of Demand Paging

Demand paging can have a significant effect on the performance of a computer system. To see why, let us compute the *effective access time* for a demand-paged memory. The memory access time, *ma*, for most computer systems now ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk, and then access the desired word.

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

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

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

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

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

In any case, we are faced with three major components of the page-fault service time:

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

The first and third tasks may be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page-switch time, on the other hand, will probably be close to 24 milliseconds. A typical hard disk has an average latency of 8 milliseconds, a seek of 15 milliseconds and a transfer time of 1 millisecond. Thus, the total paging time would be close to 25 milliseconds, including hardware and software time. Remember also that we are looking at only the device service time. If a queue of processes is waiting for the device (other processes that have caused page faults), we have to add device queueing time as we wait for the paging device to be free to service our request, increasing the time to swap even more.

If we take an average page-fault service time of 25 milliseconds and a memory access time of 100 nanoseconds, then the effective access time in nanoseconds is

$$\begin{aligned}\text{effective access time} &= (1-p) \times (100) + p \times (25 \text{ milliseconds})\\ &= (1-p) \times 100 + p \times 25{,}000{,}000\\ &= 100 + 24{,}999{,}900 \times p.\end{aligned}$$

We see then that the effective access time is directly proportional to the page-fault rate. If one access out of 1000 causes a page fault, the effective access time is 25 microseconds. The computer would be slowed down by a factor of 250 because of demand paging! If we want less than 10-percent degradation, we need

$$\begin{aligned}110 &> 100 + 25{,}000{,}000 \times p,\\ 10 &> 25{,}000{,}000 \times p,\\ p &< 0.0000004.\end{aligned}$$

That is, to keep the slowdown due to paging to a reasonable level, we can allow only less than 1 memory access out of 2,500,000 to page fault.

It is important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically.

One additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used (see Chapter 12). It is therefore possible for the system to gain better paging throughput, by copying an entire file image into the swap space at process startup, and then to perform demand paging from the swap space. Systems with limited swap space can employ such a different scheme when binary files are used. Demand pages for such files are brought directly from the file system. However, when page replacement is called for, these pages can simply be overwritten (because they are never modified) and read in from the file system again if needed. Yet another option is initially to demand pages from the file system, but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are ever read from the file system, but all subsequent paging is done from swap space. This method appears to be a good compromise; it is used in BSD UNIX.

9.4 ■ Page Replacement

In our presentation so far, the page-fault rate is not a serious problem, because each page is faulted for at most once, when it is first referenced. This representation is not strictly accurate. Consider that, if a process of 10 pages actually uses only one-half of them, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had 40 frames, we could run eight processes, rather than the four that could run if each required 10 frames (five of which were never used).

If we increase our degree of multiprogramming, we are *over-allocating* memory. If we run six processes, each of which is 10 pages in size, but actually uses only five pages, we have higher CPU utilization and throughput, with 10 frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all 10 of its pages, resulting in a need for 60 frames, when only 40 are available. Although this situation may be unlikely, it becomes much more likely as we increase the multiprogramming level, so that the average memory usage is close to the available physical memory. (In our example, why stop at a multiprogramming level of six, when we can move to a level of seven or eight?)

Over-allocating will show up in the following way. While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this is a page fault and not an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds there are *no* free frames on the free-frame list; all memory is in use (Figure 9.5).

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

We could swap out a process, freeing all its frames, and reducing the level of multiprogramming. This option is a good idea at times, and we

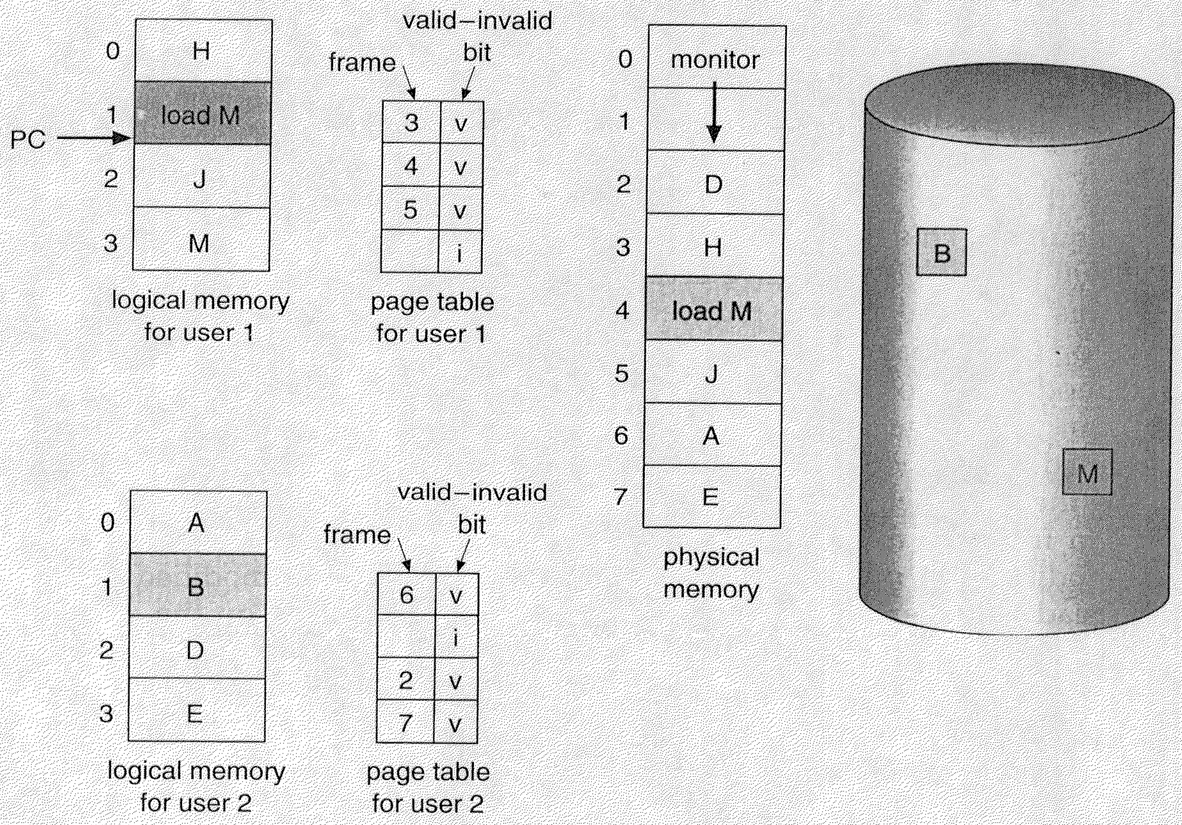


Figure 9.5 Need for page replacement.

consider it further in Section 9.7. First, we shall discuss a more intriguing possibility: *page replacement*.

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space, and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 9.6). The freed frame can now be used to hold the page for which the process faulted. The page-fault service routine is now modified to include page replacement:

1. Find the location of the desired page on the disk.

2. Find a free frame:

   a. If there is a free frame, use it.

   b. Otherwise, use a page-replacement algorithm to select a *victim* frame.

   c. Write the victim page to the disk; change the page and frame tables accordingly.

3. Read the desired page into the (newly) free frame; change the page and frame tables.
4. Restart the user process.

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

This overhead can be reduced by the use of a *modify* (*dirty*) *bit*. Each page or frame may have a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write that page to the disk. If the modify bit is not set, however, the page has *not* been modified since it was read into memory. Therefore, if the copy of the page on the disk has not been overwritten (by some other page, for example), we can avoid writing the memory page to the disk; it is already there. This technique also applies to read-only pages (for example, pages of binary code). Such

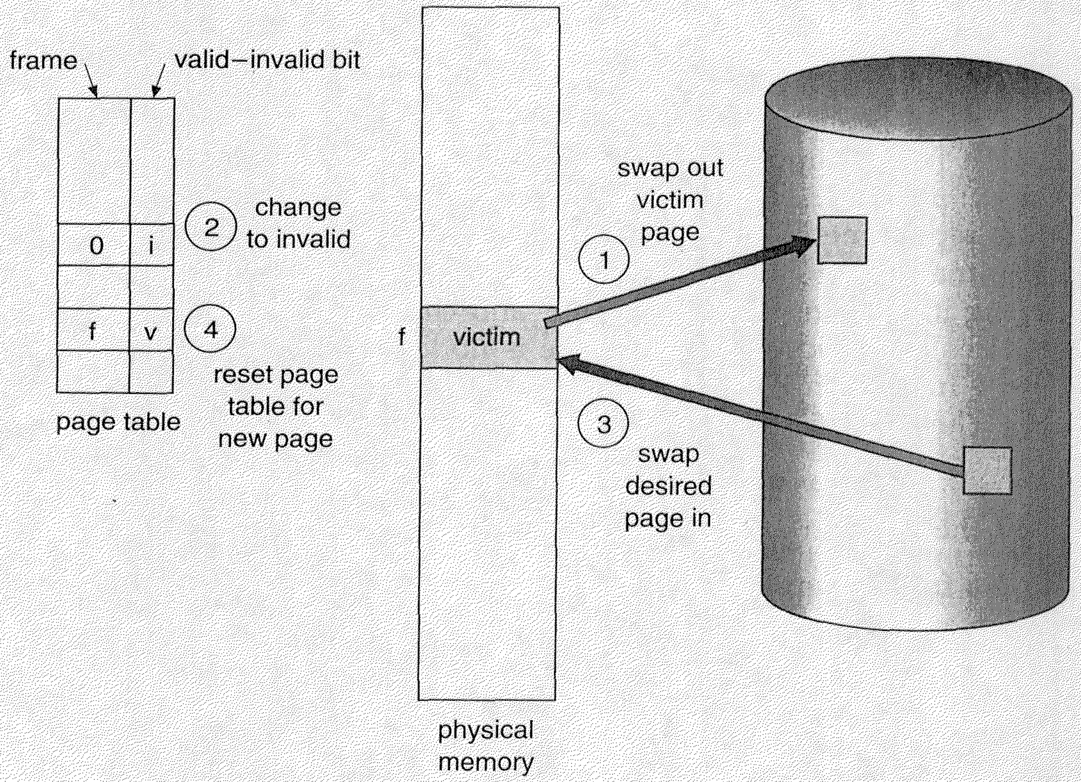


Figure 9.6 Page replacement.

pages cannot be modified; thus, they may be discarded when desired. This scheme can reduce significantly the time to service a page fault, since it reduces I/O time by one-half *if* the page is not modified.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, a very large virtual memory can be provided for programmers on a smaller physical memory. With non-demand paging, user addresses were mapped into physical addresses, allowing the two sets of addresses to be quite different. All of the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a user process of 20 pages, we can execute it in 10 frames simply by using demand paging, and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to the disk. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process.

We must solve two major problems to implement demand paging: We must develop a *frame-allocation algorithm* and a *page-replacement algorithm*. If we have multiple processes in memory, we must decide how many frames to allocate to each process. Further, when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.

9.5 ■ Page-Replacement Algorithms

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

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

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

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

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

which, at 100 bytes per page, is reduced to the following reference string

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

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults will decrease. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults, one fault for the first reference to each page. On the other hand, with only one frame available, we would have a replacement with every reference, resulting in 11 faults. In general, we expect a curve such as that in Figure 9.7. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

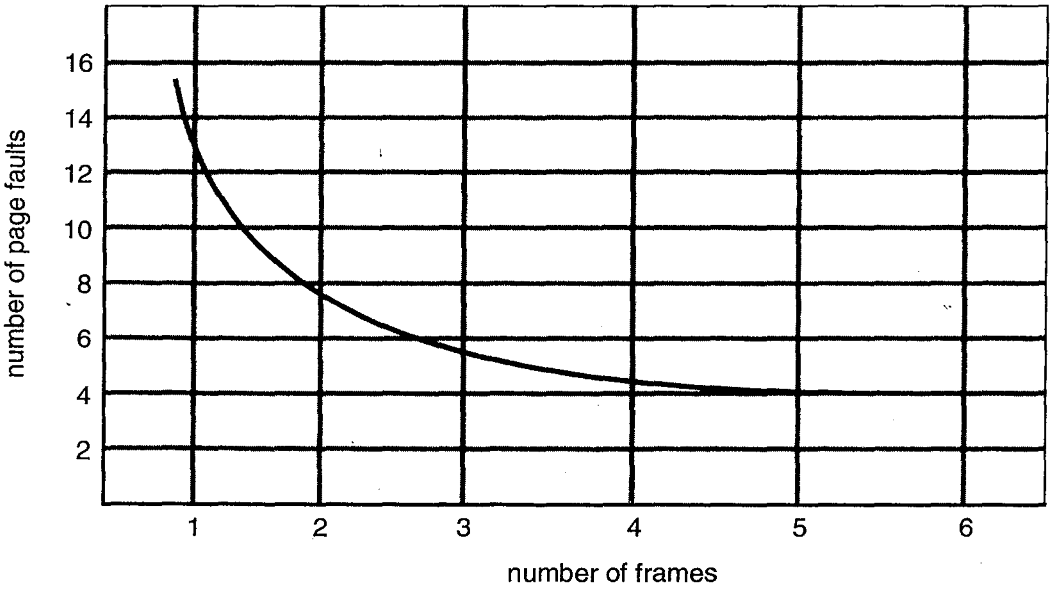


Figure 9.7 Graph of page faults versus the number of frames.

To illustrate the page-replacement algorithms, we shall use the reference string

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

for a memory with three frames.

9.5.1 FIFO Algorithm

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1, and 2) to be brought in. This replacement means that the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 9.8. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

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

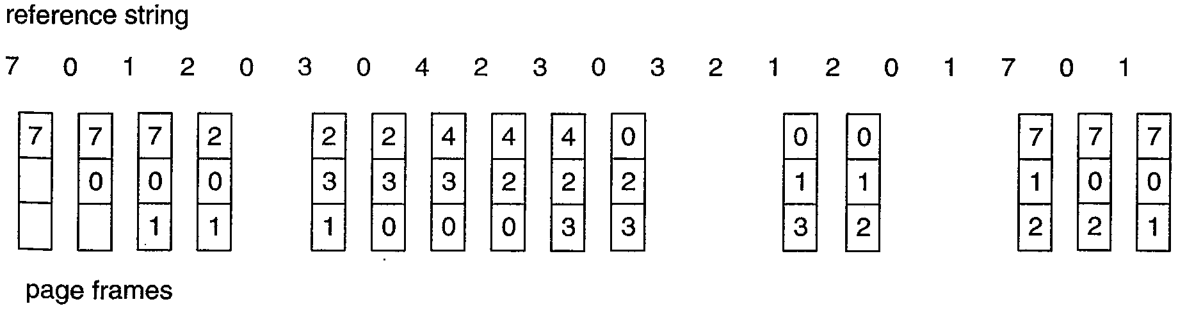


Figure 9.8 FIFO page-replacement algorithm.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution, but does not cause incorrect execution.

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

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

Figure 9.9 shows the curve of page faults versus the number of available frames. We notice that the number of faults for four frames (10) is *greater* than the number of faults for three frames (nine)! This result is most unexpected and is known as *Belady's anomaly*. Belady's anomaly reflects the fact that, for some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

9.5.2 Optimal Algorithm

One result of the discovery of Belady's anomaly was the search for an *optimal* page-replacement algorithm. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal

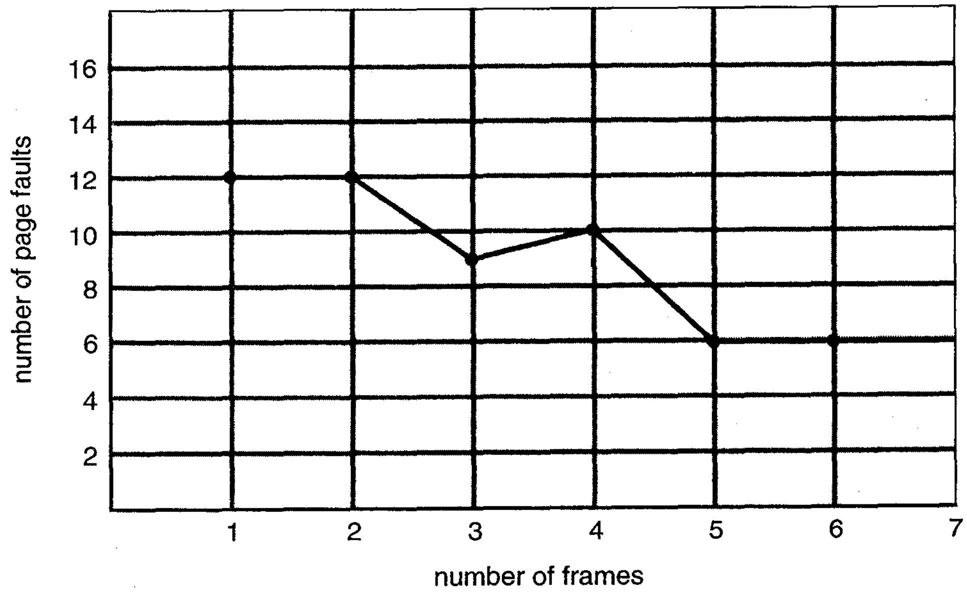


Figure 9.9 Page-fault curve for FIFO replacement on a reference string.

algorithm will never suffer from Belady's anomaly. An optimal page-replacement algorithm exists, and has been called OPT or MIN. It is simply

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

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

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 9.10. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with less than nine faults.

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

9.5.3 LRU Algorithm

If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the

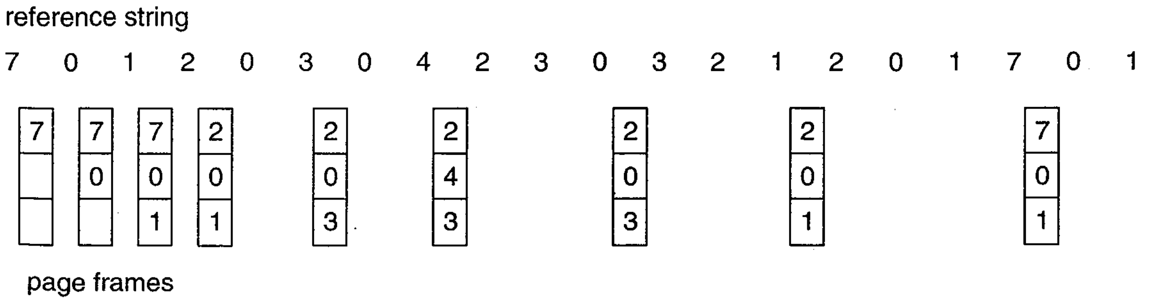


Figure 9.10 Optimal page-replacement algorithm.

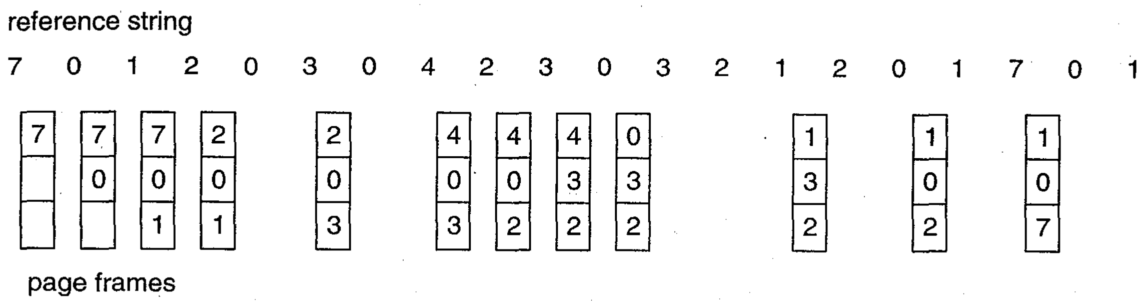


Figure 9.11 LRU page-replacement algorithm.

FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be *used.* If we use the recent past as an approximation of the near future, then we will replace the page that *has not been used* for the longest period of time (Figure 9.11). This approach is the *least recently used (LRU)* algorithm.

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

The result of applying LRU replacement to our example reference string is shown in Figure 9.11. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory {0, 3, 4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.

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

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

- **Stack:** Another approach to implementing LRU replacement is to keep a *stack* of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page (Figure 9.12). Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Neither optimal replacement nor LRU replacement suffers from Belady's anomaly. There is a class of page-replacement algorithms, called *stack algorithms,* that can never exhibit Belady's anomaly. A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for *n*

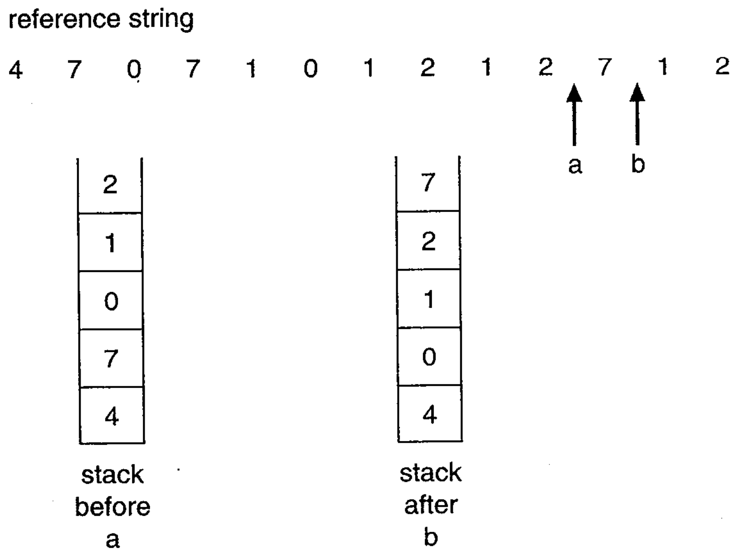


Figure 9.12 Use of a stack to record the most recent page references.

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

Note that neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers. The updating of the clock fields or stack must be done for *every* memory reference. If we were to use an interrupt for every reference, to allow software to update such data structures, it would slow every memory reference by a factor of at least 10, hence slowing every user process by a factor of 10. Few systems could tolerate that level of overhead for memory management.

9.5.4 LRU Approximation Algorithms

Few systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a *reference bit*. The reference bit for a page is set, by the hardware, whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

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

9.5.4.1 Additional-Reference-Bits Algorithm

We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right 1 bit, discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

A page with a history register value of 11000100 has been used more recently than has one with 01110111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value, or use a FIFO selection among them.

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

9.5.4.2 Second-Chance Algorithm

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

One way to implement the second-chance (sometimes referred to as the clock) algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 9.13). Once a victim page is found, the page is replaced and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.

9.5.4.3 Enhanced Second-Chance Algorithm

The second-chance algorithm described above can be enhanced by considering both the reference bit and the modify bit (Section 9.4) as an ordered pair. With these 2 bits, we have the following four possible classes:

1. (0,0) neither recently used nor modified - best page to replace

2. (0,1) not recently used but modified - not quite as good, because the page will need to be written out before replacement

3. (1,0) recently used but clean - probably will be used again soon

4. (1,1) recently used and modified - probably will be used again, and write out will be needed before replacing it

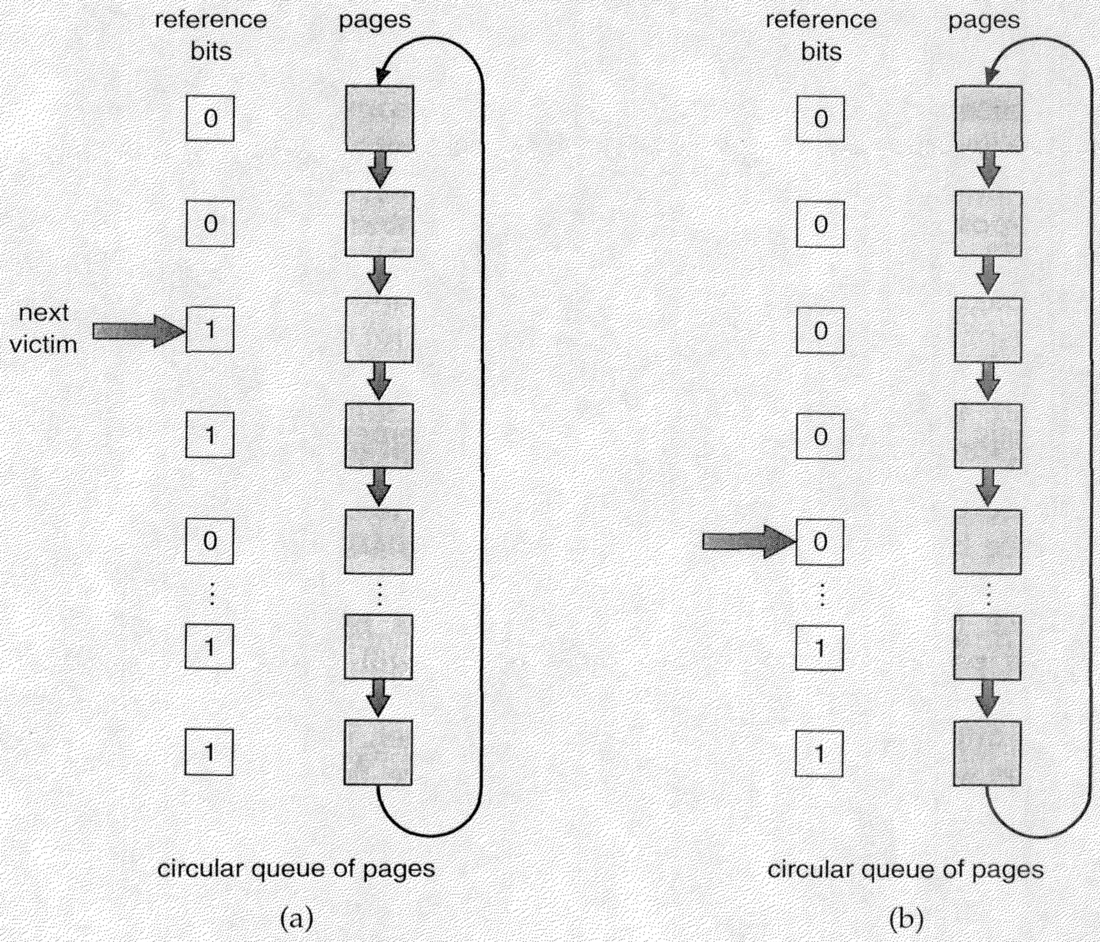


Figure 9.13 Second-chance (clock) page-replacement algorithm.

When page replacement is called for, each page is in one of these four classes. We use the same scheme as the clock algorithm, but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced.

This algorithm is used in the Macintosh virtual-memory-management scheme. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of I/Os required.

9.5.5 Counting Algorithms

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

- LFU **Algorithm:** The *least frequently used (LFU)* page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.
- MFU **Algorithm:** The *most frequently used (MFU)* page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

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

9.5.6 Page Buffering Algorithm

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

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

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

This technique is used in the VAX/VMS system, with a FIFO replacement algorithm. When the FIFO replacement algorithm mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm. This method is necessary because the early versions of the VAX did not correctly implement the reference bit.

9.6 ■ Allocation of Frames

How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?

The simplest case of virtual memory is the single-user system. Consider a single-user microcomputer system with 128K memory composed of pages of size 1K. Thus, there are 128 frames. The operating system may take 35K, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the ninety-fourth, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

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

Other variants are also possible, but the basic strategy is clear: The user process is allocated any free frame.

A different problem arises when demand paging is combined with multiprogramming. Multiprogramming puts two (or more) processes in memory at the same time.

9.6.1 Minimum Number of Frames

There are, of course, various constraints on our strategies for the allocations of frames. We cannot allocate more than the total number of available frames (unless there is page sharing). There is also a minimum number of frames that can be allocated. Obviously, as the number of frames allocated to each process decreases, the page fault-rate increases, slowing process execution.

Besides the undesirable performance properties of allocating only a few frames, there is a minimum number of frames that must be allocated. This minimum number is defined by the instruction-set architecture. Remember that, when a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently, we must have enough frames to hold all the different pages that any single instruction can reference.

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

The minimum number of frames is defined by the computer architecture. For example, the move instruction for the PDP-11 is more than one word for some addressing modes, and thus the instruction itself may straddle two pages. In addition, each of its two operands may be indirect references, for a total of six frames. The worst case for the IBM 370 is probably the MVC instruction. Since the instruction is storage to storage, it takes 6 bytes and can straddle two pages. The block of characters to move and the area to be moved to can each also straddle two pages. This situation would require six frames. (Actually, the worst case is if the MVC instruction is the operand of an EXECUTE instruction that straddles a page boundary; in this case, we need eight frames.)

The worst-case scenario occurs in architectures that allow multiple levels of indirection (for example, each 16-bit word could contain a 15-bit address plus a 1-bit indirect indicator). Theoretically, a simple load instruction could reference an indirect address that could reference an indirect address (on another page) that could also reference an indirect address (on yet another page), and so on, until every page in virtual memory had been touched. Thus, in the worst case, the entire virtual memory must be in physical memory. To overcome this difficulty, we must place a limit on the levels of indirection (for example, limit an instruction to at most 16 levels of indirection). When the first indirection occurs, a counter is set to 16; the counter is then decremented for each successive indirection for this instruction. If the counter is decremented to 0, a trap occurs (excessive indirection). This limitation reduces the maximum number of memory references per instruction to 17, requiring the same number of frames.

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

9.6.2 Allocation Algorithms

The easiest way to split *m* frames among *n* processes is to give everyone an equal share, *m*/*n* frames. For instance, if there are 93 frames and five processes, each process will get 18 frames. The leftover three frames could be used as a free-frame buffer pool. This scheme is called *equal allocation.*

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

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

$$S = \Sigma\, s_i.$$

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

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

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

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

$$10/137 \times 62 \simeq 4,$$
$$127/137 \times 62 \simeq 57.$$

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

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

Notice that, with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes.

One approach is to use a proportional allocation scheme where the ratio of frames depends not on the relative sizes of processes, but rather on the processes' priorities, or on a combination of size and priority.

9.6.3 Global Versus Local Allocation

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

For example, consider an allocation scheme where we allow high-priority processes to select frames from low-priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of the low-priority process.

With a local replacement strategy, the number of frames allocated to a process does not change. With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes do not choose *its* frames for replacement).

One problem with a global replacement algorithm is that a process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behavior of that process, but also on the paging behavior of other processes. Therefore, the same process may perform quite differently (taking 0.5 seconds for one execution and 10.3 seconds for the next execution) due to totally external circumstances. Such is not the case with a local replacement algorithm. Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. For its part, local replacement might hinder a process by not making available to it other, less used pages of memory. Thus, global replacement generally results in greater system throughput, and is therefore the more common method.

9.7 ■ Thrashing

If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process' execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.

In fact, look at any process that does not have "enough" frames. Although it is technically possible to reduce the number of allocated frames to the minimum, there is some (larger) number of pages that are in active use. If the process does not have this number of frames, it will very quickly page fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it very quickly faults again, and again, and again. The process continues to fault, replacing pages for which it will then fault and bring back in right away.

This high paging activity is called *thrashing*. A process is thrashing if it is spending more time paging than executing.

9.7.1 Cause of Thrashing

Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behavior of early paging systems.

The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used, replacing pages with no regard to the process to which they belong. Now suppose a process enters a new phase in its execution and needs more frames. It starts faulting and taking pages away from other processes. These processes need those pages, however, and so they also fault, taking pages from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

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

This phenomenon is illustrated in Figure 9.14. CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.

The effects of thrashing can be limited by using a *local (or priority) replacement algorithm.* With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash also. Pages are replaced with regard to the process of which they are a part. However, if processes are thrashing, they will be in the queue

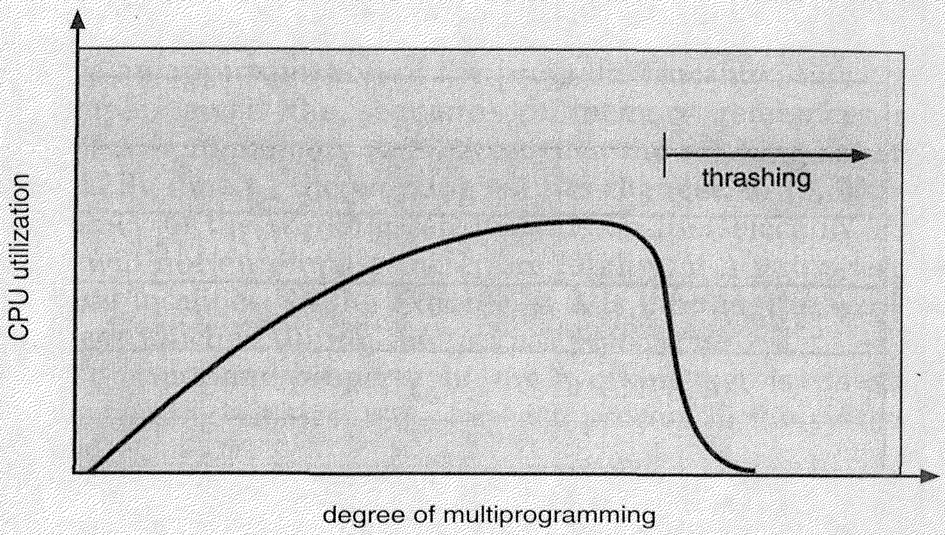


Figure 9.14 Thrashing.

for the paging device most of the time. The average service time for a page fault will increase, due to the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

To prevent thrashing, we must provide a process as many frames as it needs. But how do we know how many frames it "needs"? There are several techniques. The working-set strategy (discussed in Section 9.7.2) starts by looking at how many frames a process is actually using. This approach defines the *locality model* of process execution.

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together (Figure 9.15). A program is generally composed of several different localities, which may overlap.

For example, when a subroutine is called, it defines a new locality. In this locality, memory references are made to the instructions of the subroutine, its local variables, and a subset of the global variables. When the subroutine is exited, the process leaves this locality, since the local variables and instructions of the subroutine are no longer in active use. We may return to this locality later. Thus, we see that localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure. Note that the locality model is the unstated principle behind the caching discussions so far in this book. If accesses to any types of data were random rather than patterned, caching would be useless.

Suppose that we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities.

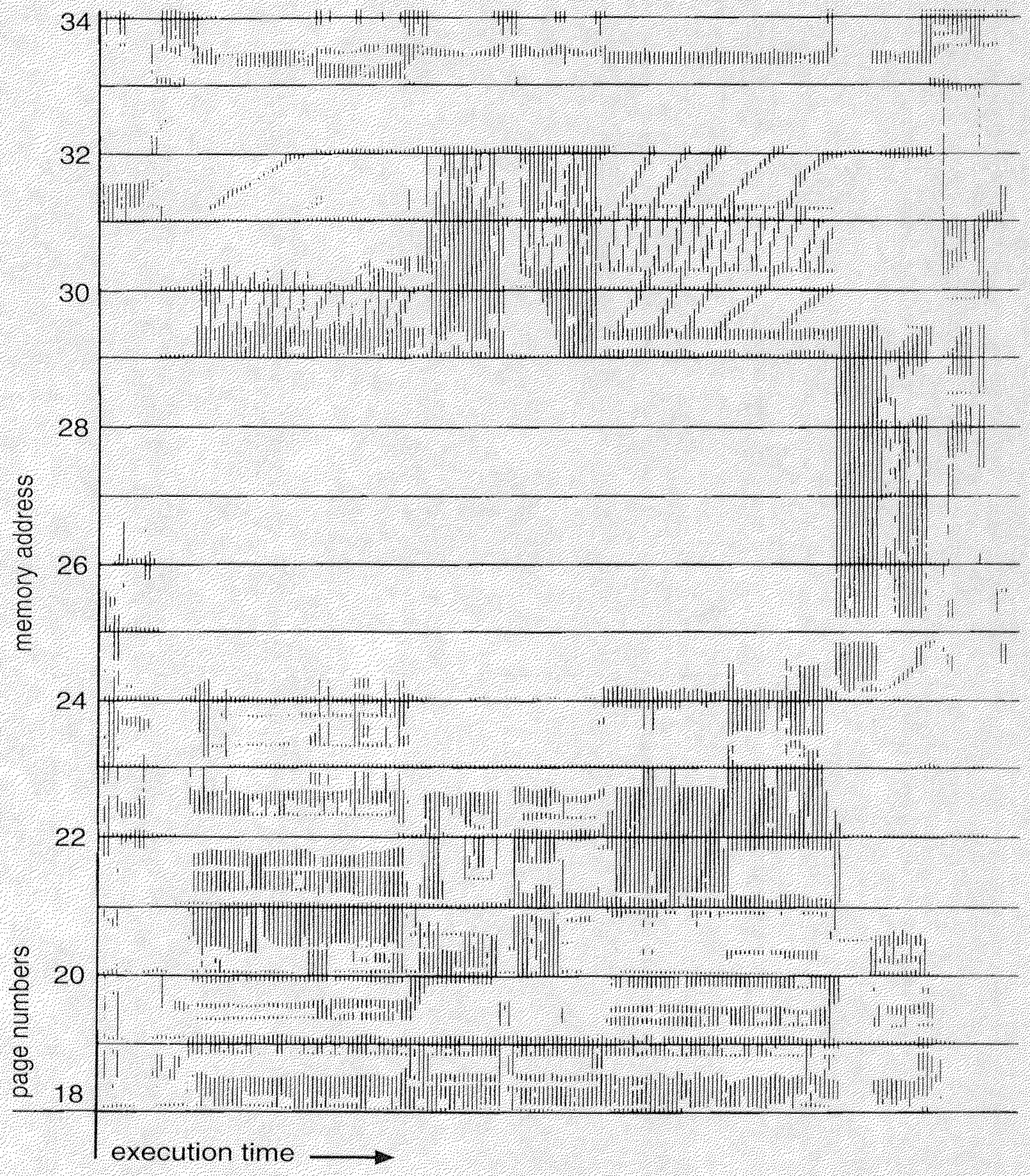


Figure 9.15 Locality in a memory reference pattern.

If we allocate fewer frames than the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

9.7.2 Working-Set Model

The *working-set model* is based on the assumption of locality. This model uses a parameter, Δ, to define the *working-set window*. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is the *working set* (Figure 9.16). If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. Thus, the working set is an approximation of the program's locality.

For example, given the sequence of memory references shown in Figure 9.16, if $\Delta = 10$ memory references, then the working set at time t_1 is $\{1, 2, 5, 6, 7\}$. By time t_2, the working set has changed to $\{3, 4\}$.

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

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

$$D = \Sigma \ WSS_i,$$

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

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

This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization.

The difficulty with the working-set model is keeping track of the working set. The working-set window is a moving window. At each memory reference, a new reference appears at one end and the oldest reference drops off the other end. A page is in the working set if it is referenced anywhere in the working-set window. We can approximate the working-set model with a fixed interval timer interrupt and a reference bit.

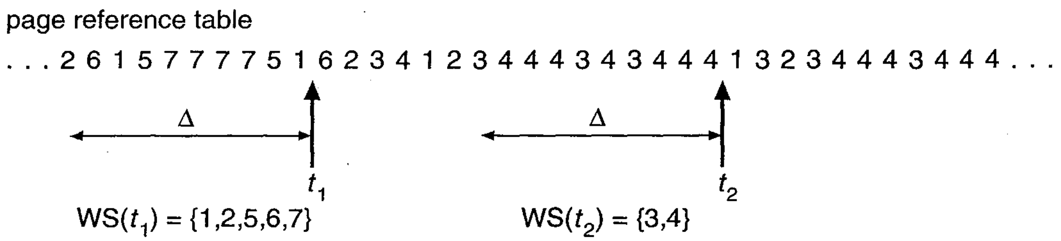


Figure 9.16 Working-set model.

For example, assume Δ is 10,000 references and we can cause a timer interrupt every 5000 references. When we get a timer interrupt, we copy and clear the reference-bit values for each page. Thus, if a page fault occurs, we can examine the current reference bit and 2 in-memory bits to determine whether a page was used within the last 10,000 to 15,000 references. If it was used, at least 1 of these bits will be on. If it has not been used, these bits will be off. Those pages with at least 1 bit on will be considered to be in the working set. Note that this arrangement is not entirely accurate, because we cannot tell where, within an interval of 5000, a reference occurred. We can reduce the uncertainty by increasing the number of our history bits and the number of interrupts (for example, 10 bits and interrupts every 1000 references). However, the cost to service these more frequent interrupts will be correspondingly higher.

9.7.3 Page-Fault Frequency

The working-set model is successful, and knowledge of the working set can be useful for prepaging (Section 9.8.1), but it seems a clumsy way to control thrashing. The *page-fault frequency (PFF)* strategy takes a more direct approach.

The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Similarly, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate (Figure 9.17). If the actual page-fault rate exceeds the upper limit, we allocate that process another frame; if the page-fault rate falls below the lower limit, we remove a frame from that process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

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

9.8 ■ Other Considerations

The selections of a replacement algorithm and allocation policy are the major decisions to make for a paging system. There are many other considerations as well.

9.8.1 Prepaging

An obvious property of a pure demand-paging system is the large number of page faults that occur when a process is started. This situation is a result

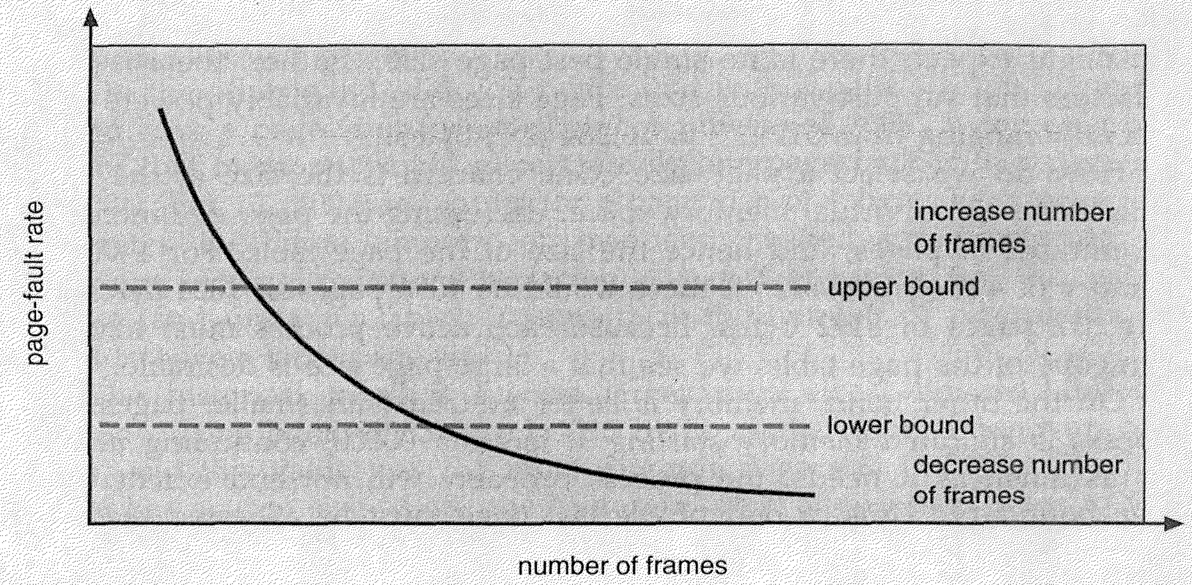


Figure 9.17 Page-fault frequency.

of trying to get the initial locality into memory. The same thing may happen at other times. For instance, when a swapped-out process is restarted, all its pages are on the disk and each must be brought in by its own page fault. *Prepaging* is an attempt to prevent this high level of initial paging. The strategy is to bring into memory at one time all the pages that will be needed.

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

Prepaging may be an advantage in some cases. The question is simply whether the cost of prepaging is less than the cost of servicing the corresponding page faults. It may well be the case that many of the pages brought back into memory by prepaging are not used. Assume that s pages are prepaged and a fraction α of these s pages are actually used ($0 \leq \alpha \leq 1$). The question is whether the cost of the α saved page faults is greater or less than the cost of prepaging ($1 - \alpha$) unnecessary pages. If α is close to zero, prepaging loses; if α is close to one, prepaging wins.

9.8.2 Page Size

The designers of an operating system for an existing machine seldom have a choice concerning the page size. However, when new machines are being designed, a decision regarding the best page size must be made. As you might expect, there is no single best page size. Rather, there is a set of factors that support various sizes. Page sizes are invariably powers of 2, generally ranging from 512 (2^9) to 16,384 (2^{14}) bytes.

How do we select a page size? One concern is the size of the page table. For a given virtual memory space, decreasing the page size increases the number of pages, and hence the size of the page table. For a virtual memory of 4 megabytes (2^{22}), there would be 4096 pages of 1024 bytes but only 512 pages of 8192 bytes. Because each active process must have its own copy of the page table, we see that a large page size is desirable.

On the other hand, memory is better utilized with smaller pages. If a process is allocated memory starting at location 00000, continuing until it has as much as it needs, the process probably will not end exactly on a page boundary. Thus, a part of the last page must be allocated (because pages are the units of allocation) but is unused (internal fragmentation). Assuming independence of process size and page size, we would expect that, on the average, one-half of the last page of each process will be wasted. This loss would be only 256 bytes for a page of 512 bytes, but would be 4096 bytes for a page of 8192 bytes. To minimize internal fragmentation, we need a small page size.

Another problem is the time required to read or write a page. I/O time is composed of seek, latency, and transfer times. Transfer time is proportional to the amount transferred (that is, the page size), a fact that would seem to argue for a small page size. Remember from Chapter 2, however, that latency and seek time normally dwarf transfer time. At a transfer rate of 2 megabytes per second, it takes only 0.2 milliseconds to transfer 512 bytes. Latency, on the other hand, is perhaps 8 milliseconds and seek time 20 milliseconds. Of the total I/O time (28.2 milliseconds), therefore, 1 percent is attributable to the actual transfer. Doubling the page size increases I/O time to only 28.4 milliseconds. It takes 28.4 milliseconds to read a single page of 1024 bytes, but 56.4 milliseconds to read the same amount as two pages of 512 bytes each. Thus, a desire to minimize I/O time argues for a larger page size.

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

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

The historical trend is toward larger page sizes. Indeed, the first edition of this book (1983) used 4096 bytes as the upper bound on page sizes, and this value was the most common page size in 1990. The Intel 80386 has a page size of 4K; the Motorola 68030 allows page sizes to vary from 256 bytes to 32K. The evolution to larger page sizes is probably the result of CPU speeds and main memory capacity increasing faster than have disk speeds. Page faults are more costly today, in overall system performance, than previously. It is therefore advantageous to increase page sizes to reduce their frequency. Of course, there is more internal fragmentation as a result.

There are other factors to consider (such as the relationship between page size and sector size on the paging device). The problem has no best answer. Some factors (internal fragmentation, locality) argue for a small page size, whereas others (table size, I/O time) argue for a large page size. At least two systems allow two different page sizes. The MULTICS hardware (GE 645) allows pages of either 64 words or 1024 words. The IBM/370 allows pages of either 2K or 4K. The difficulty of picking a page size is illustrated by the fact that MVS on the IBM/370 selected 4K pages, whereas VS/1 selected 2K pages.

9.8.3 Program Structure

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

As a contrived but informative example, assume pages are 128 words in size. Consider a Pascal program whose function is to initialize to 0 each element of a 128 by 128 array. The following code is typical:

```
var A: array [1..128] of array [1..128] of integer;
   for j := 1 to 128
      do for i := 1 to 128
         do A[i][j] := 0;
```

Notice that the array is stored row major. That is, the array is stored *A*[1][1], *A*[1][2], ..., *A*[1][128], *A*[2][1], *A*[2][2], ..., *A*[128][128]. For pages of 128 words, each row takes one page. Thus, the proceeding code zeros one word in each page, then another word in each page, and so on. If the operating system allocates less than 128 frames to the entire program, then its execution will result in 128 × 128 = 16,384 page faults. Changing the code to

```
var A: array [1..128] of array [1..128] of integer;
    for i := 1 to 128
       do for j := 1 to 128
          do A[i][j] := 0;
```

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

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

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

The choice of programming language can affect paging as well. For example, LISP uses pointers frequently, and pointers tend to randomize access to memory. Contrast LISP with PASCAL, which uses few pointers. PASCAL programs will have better locality of reference and therefore generally will execute faster than LISP programs on a virtual memory system.

9.8.4 I/O Interlock

When demand paging is used, we sometimes need to allow some of the pages to be *locked* in memory. One such situation occurs when I/O is done to or from user (virtual) memory. I/O is often implemented by a separate I/O processor. For example, a magnetic-tape controller is generally given the number of bytes to transfer and a memory address for the buffer (Figure 9.18). When the transfer is complete, the CPU is interrupted.

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

There are two common solutions to this problem. One solution is never to execute I/O to user memory. Instead, data are always copied between system memory and user memory. I/O takes place only between system memory and the I/O device. To write a block on tape, we first copy the block to system memory, and then write it to tape.

This extra copying may result in unacceptably high overhead. Another solution is to allow pages to be *locked* into memory. A lock bit is associated with every frame. If the frame is locked, it cannot be selected for replacement. Under this approach, to write a block on tape, we lock into memory the pages containing the block. The system can then continue as

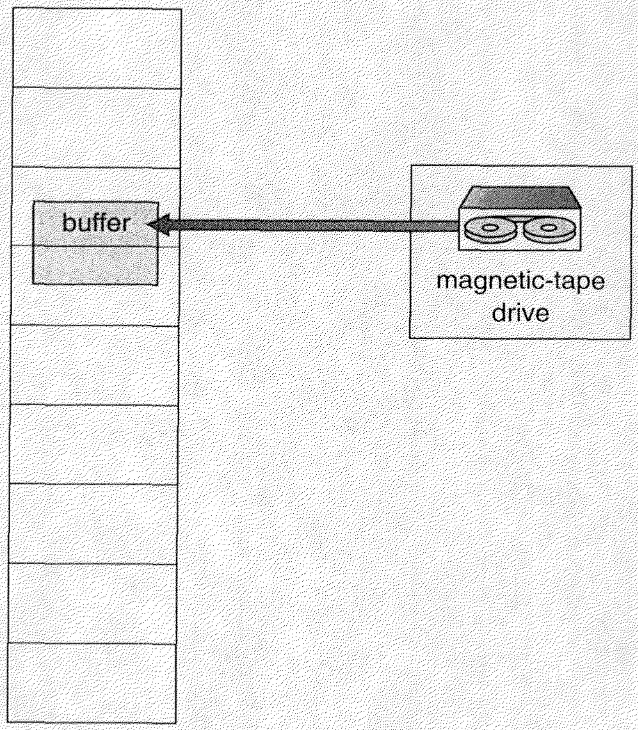


Figure 9.18 Diagram showing why frames used for I/O must be in memory.

usual. Locked pages cannot be replaced. When the I/O is complete, the pages are unlocked.

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

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

Using a lock bit can be dangerous, however, if it gets turned on but never gets turned off. Should this situation occur (due to a bug in the operating system, for example), the locked frame becomes unusable. The Macintosh Operating System provides a page locking mechanism because it is a single-user system, and the overuse of locking would only hurt the user doing the locking. Multi-user systems need to be less trusting of users. For instance, SunOS allows locking "hints," but is free to disregard them if the free page pool becomes too small or if an individual process is requesting too many pages be locked in memory.

9.8.5 Real-Time Processing

The discussions in this chapter have concentrated on providing the best overall utilization of a computer system by optimizing the use of memory. By using memory for active data, and moving inactive data to disk, we increase overall system throughput. However, individual processes may suffer as a result, because they now may take additional page faults during their execution.

Consider a real-time process or thread, as described in Chapter 4. Such a process expects to gain control of the CPU, and to run to completion with a minimum of delays. Virtual memory is the antithesis of real-time computing, because it can introduce unexpected, long-term delays in the execution of a process while pages are brought into memory. Therefore, real-time systems almost never have virtual memory.

In the case of Solaris 2, Sun wanted to allow both time-sharing and real-time computing within a system. To solve the page fault problem, Solaris 2 allows a process to tell the system which pages are important to that process. In addition to allowing "hints" on page use that we have mentioned, the operating system allows privileged users to require pages to be locked into memory. If abused, this mechanism could lock all other processes out of the system. It is necessary to allow real-time processes to have bounded and low dispatch latency.

9.9 ■ Demand Segmentation

Although demand paging is generally considered the most efficient virtual-memory system, a significant amount of hardware is required to implement it. When this hardware is lacking, less efficient means are sometimes devised to provide virtual memory. A case in point is *demand segmentation.* The Intel 80286 does not include paging features, but does have segments. The OS/2 operating system, which runs on this CPU, uses the segmentation hardware to implement demand segmentation as the only possible approximation of demand paging.

OS/2 allocates memory in segments, rather than in pages. It keeps track of these segments through *segment descriptors,* which include information about the segment's size, protections, and location. A process does not need to have all its segments in memory to execute. Instead, the segment descriptor contains a valid bit for each segment to indicate whether the segment is currently in memory. When a process addresses a segment containing either code or data, the hardware checks this valid bit. If the segment is in main memory, the access continues unhindered. If the segment is not in memory, a trap to the operating system occurs (segment fault), just as in demand-paging implementations. OS/2 then swaps out a segment to secondary storage, and brings in the entire requested segment. The interrupted instruction (the one causing the segment fault) then continues.

To determine which segment to replace in case of a segment fault, OS/2 uses another bit in the segment descriptor called *accessed bit.* An accessed bit serves the same purpose as does a reference bit in a demand-paging environment. It is set whenever any byte in the segment is either read or written. A queue is kept containing an entry for each segment in memory. After every time slice, the operating system places at the head of the queue any segments with a set access bit. It then clears all access bits. In this way, the queue stays ordered with the most recently used segments at the head. In addition, OS/2 provides system calls that processes can use to inform the system of those segments that can either be discarded, or must always remain in memory. This information is used to rearrange the entries in the queue. When an invalid-segment trap occurs, the memory-management routines first determine whether there is sufficient free memory space to accommodate the segment. Memory compaction may be done to get rid of external fragmentation. If, after compaction, there is still not sufficient free memory, segment replacement is performed. The segment at the end of the queue is chosen for replacement and is written to swap space. If the newly freed space is large enough to accommodate the requested segment, then the requested segment is read into the vacated segment, the segment descriptor is updated, and the segment is placed at the head of the queue. Otherwise, memory compaction is performed, and the procedure is repeated.

It should be clear that demand segmentation requires considerable overhead. Thus, demand segmentation is not an optimal means for making best use of the resources of a computer system. The alternative on less sophisticated hardware is, however, no virtual memory at all. Given the problems entailed in systems lacking virtual memory, such as those described in Chapter 8, that solution is also lacking. Demand segmentation is therefore a reasonable compromise of functionality given hardware constraints that make demand paging impossible.

9.10 ■ Summary

It is desirable to be able to execute a process whose logical address space is larger than the available physical address space. The programmer can make such a process executable by restructuring it using overlays, but this is generally a difficult programming task. Virtual memory is a technique to allow a large logical address space to be mapped onto a smaller physical memory. Virtual memory allows extremely large processes to be run, and also allows the degree of multiprogramming to be raised, increasing CPU utilization. Further, it frees application programmers from worrying about memory availability.

Pure demand paging never brings in a page until that page is actually referenced. The first reference causes a page fault to the operating-system resident monitor. The operating system consults an internal table to determine where the page is located on the backing store. It then finds a free frame and reads the page in from the backing store. The page table is updated to reflect this change, and the instruction that caused the page fault is restarted. This approach allows a process to run even though its entire memory image is not in main memory at once. As long as the page-fault rate is reasonably low, performance is acceptable.

Demand paging can be used to reduce the number of frames allocated to a process. This arrangement can raise the degree of multiprogramming (allowing more processes to be available for execution at one time) and —

in theory, at least — the CPU utilization of the system. It also allows processes to be run even though their memory requirements exceed the total available physical memory. Such processes run in virtual memory.

If total memory requirements exceed the physical memory, then it may be necessary to replace pages from memory to free frames for new pages. Various page-replacement algorithms are used. FIFO page replacement is easy to program, but suffers from Belady's anomaly. Optimal page replacement requires future knowledge. LRU replacement is an approximation of optimal, but even it may be difficult to implement. Most page-replacement algorithms, such as the second-chance algorithm, are approximations of LRU replacement.

In addition to a page-replacement algorithm, a frame-allocation policy is needed. Allocation can be fixed, suggesting local page replacement, or dynamic, suggesting global replacement. The working-set model assumes that processes execute in localities. The working set is the set of pages in the current locality. Accordingly, each process should be allocated enough frames for its current working set.

If a process does not have enough memory for its working set, it will thrash. Providing enough frames to each process to avoid thrashing may require process swapping and scheduling.

In addition to requiring that we solve the major problems of page replacement and frame allocation, the proper design of a paging system requires that we consider page size, I/O, locking, prepaging, program structure, and other topics. Virtual memory can be thought of as one level of a hierarchy of storage levels in a computer system. Each level has its own access time, size, and cost parameters. A full example of a hybrid, functional virtual-memory system is presented in Chapter 20.

■ Exercises

9.1 When do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

9.2 Assume you have a page reference string for a process with m frames (initially all empty). The page reference string has length p with n distinct page numbers occurring in it. For any page-replacement algorithms,

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

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

9.3 A certain computer provides its users with a virtual-memory space of 2^{32} bytes. The computer has 2^{18} bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.

9.4 Which of the following programming techniques and structures are "good" for a demand-paged environment ? Which are "not good"? Explain your answers.

a. Stack

b. Hashed symbol table

c. Sequential search

d. Binary search

e. Pure code

f. Vector operations

g. Indirection

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

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

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

a. LRU replacement

b. FIFO replacement

c. Optimal replacement

d. Second-chance replacement

9.7 When virtual memory is implemented in a computing system, there are certain costs associated with the technique, and certain benefits. List the costs and the benefits. Is it possible for the costs to exceed the benefits? If it is, what measures can be taken to ensure that this does not happen?

9.8 An operating system supports a paged virtual memory, using a central processor with a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1000 words, and the paging device is a drum that rotates at

3000 revolutions per minute, and transfers 1 million words per second. The following statistical measurements were obtained from the system:

- 1 percent of all instructions executed accessed a page other than the current page.
- Of the instructions that accessed another page, 80 percent accessed a page already in memory.
- When a new page was required, the replaced page was modified 50 percent of the time.

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

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

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

Which (if any) of the following will (probably) improve CPU utilization? Explain your answer.

a. Install a faster CPU.

b. Install a bigger paging disk.

c. Increase the degree of multiprogramming.

d. Decrease the degree of multiprogramming.

e. Install more main memory.

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

g. Add prepaging to the page fetch algorithms.

h. Increase the page size.

9.10 Consider the two-dimensional array A:

var A: **array** [1..100] **of array** [1..100] **of** *integer;*

where $A[1][1]$ is at location 200, in a paged memory system with pages of size 200. A small process is in page 0 (locations 0 to 199) for manipulating the matrix; thus, every instruction fetch will be from page 0.

For three page frames, how many page faults are generated by the following array-initialization loops, using LRU replacement, and assuming page frame 1 has the process in it, and the other two are initially empty:

a.
```
for j:= 1 to 100 do
    for i := 1 to 100 do
        A[i][j]:= 0;
```

b.
```
for i := 1 to 100 do
    for j:= 1 to 100 do
        A[i][j]:= 0;
```

9.11 Consider the following page reference string:

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

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

- LRU replacement
- FIFO replacement
- Optimal replacement

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

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

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

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

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

a. Define a page-replacement algorithm using this basic idea. Specifically address the problems of (1) What the initial value of the counters is, (2) when counters are increased, (3) when counters are decreased, and (4) how the page to be replaced is selected.

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

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

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

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

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

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

a. CPU utilization 13 percent; disk utilization 97 percent

b. CPU utilization 87 percent; disk utilization 3 percent

c. CPU utilization 13 percent; disk utilization 3 percent

9.19 We have an operating system for a machine that uses base and limit registers, but we have modified the machine to provide a page table.

Can the page tables be set up to simulate base and limit registers? How can they be, or why can they not be?

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

Bibliographic Notes

Demand paging was first used in the Atlas system, implemented on the Manchester University MUSE computer around 1960 [Kilburn et al. 1961]. Another early demand-paging system was MULTICS, implemented on the GE 645 system [Organick 1972].

Belady et al. [1969] were the first researchers to observe that the FIFO replacement strategy may have the anomaly that bears Belady's name. Mattson et al. [1970] demonstrated that stack algorithms are not subject to Belady's anomaly.

The optimal replacement algorithm was presented by Belady [1966]. It was proved to be optimal by Mattson et al. [1970]. Belady's optimal algorithm is for a fixed allocation; Prieve and Fabry [1976] have an optimal algorithm for situations where the allocation can vary.

The enhanced clock algorithm was discussed by Carr and Hennessy [1981]; it is used in the Macintosh virtual memory-management scheme, and was described by Goldman [1989].

Thrashing was discussed by Denning [1968]. The working-set model was developed by Denning [1968]. Discussions concerning the working-set model were presented by Denning [1980].

The page-fault-rate monitoring scheme was developed by Wulf [1969], who successfully applied this technique to the Burroughs B5500 computer system. Chu and Opderbeck [1976] discussed program behavior and the page-fault-frequency replacement algorithm. Gupta and Franklin [1978] provided a performance comparison between the working-set scheme and the page-fault-frequency replacement scheme.

Demand segmentation and details of OS/2 were described by Iacobucci [1988]. Further information about OS/2 was presented in [Microsoft 1989].

The Intel 80386 paging hardware was described in [Intel 1986]; the Motorola 68030 hardware was covered in [Motorola 1989b]. Virtual-memory management in the VAX/VMS operating system was discussed by Levy and Lipman [1982]. Discussions concerning workstation operating systems and virtual memory are offered by Hagmann [1989].

CHAPTER 10

FILE SYSTEM INTERFACE

For most users, the file system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and access to both data and programs belonging to the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of *files*, each storing related data, and a *directory structure*, which organizes and provides information about all the files in the system. Some file systems have a third part, *partitions*, which are used to separate physically or logically large collections of directories. In this chapter, we consider the various aspects of files, and the variety of directory structures. We also discuss ways to handle *file protection*, which is necessary in an environment where multiple users have access to files, and where it is usually desirable to control by whom and in what ways files may be accessed. Finally, we discuss the semantics of sharing files among multiple processes.

10.1 ■ File Concept

Computers can store information on several different storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the *file*. Files are mapped, by the operating system, onto physical devices. These storage devices are usually *nonvolatile*, so the contents are persistent through power failures and system reboots.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records whose meaning is defined by the file's creator and user. The concept of a file is thus extremely general.

The information in a file is defined by its creator. Many different types of information may be stored in a file: source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined *structure* according to its type. A *text* file is a sequence of characters organized into lines (and possibly pages); a *source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements; an *object* file is a sequence of bytes organized into blocks understandable by the system's linker; an *executable* file is a series of code sections that the loader can bring into memory and execute. The internal structure of files is discussed in Section 10.1.4.

10.1.1 File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as "example.c". Some systems differentiate between upper- and lower-case characters in names, whereas other systems consider the two cases to be equivalent. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create file "example.c", whereas another user might edit that file by specifying its name. The file's owner might write the file to a floppy disk or magnetic tape, and might read it off onto another system, where it could still be called "example.c".

A file has certain other attributes, which vary from one operating system to another, but typically consist of these:

- **Name**. The symbolic file name is the only information kept in human-readable form.

- **Type**. This information is needed for those systems that support different types.

- **Location**. This information is a pointer to a device and to the location of the file on that device.

- **Size**. The current size of the file (in bytes, words or blocks), and possibly the maximum allowed size are included in this attribute.

- **Protection**. Access-control information controls who can do reading, writing, executing, and so on.

- **Time, date, and user identification**. This information may be kept for (1) creation, (2) last modification, and (3) last use. These data can be useful for protection, security, and usage monitoring.

The information about all files is kept in the directory structure that also resides on secondary storage. It may take from 16 to over 1000 bytes to record this information for each file. In a system with many files, the size of the directory itself may be megabytes. Because directories, like files, must be nonvolatile, they must be stored on the device and brought into memory piecemeal, as needed. We shall discuss the directory-structure organization in Section 10.3

10.1.2 File Operations

A file is an *abstract data type*. To define a file properly, we need to consider the operations that can be performed on files. The operating system provides system calls to create, write, read, reposition, delete, and truncate files. Let us consider what the operating system must do for each of the six basic file operations. It should then be easy to see how similar operations, such as renaming a file, would be implemented.

- **Creating a file**. Two steps are necessary to create a file. First, space in the file system must be found for the file. We shall discuss how to allocate space for the file in Chapter 11. Second, an entry for the new file must be made in the directory. The directory entry records the name of the file and the location in the file system.

- **Writing a file**. To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the location of the file. The system must keep a *write* pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

- **Reading a file**. To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated directory entry, and the system needs to keep a *read* pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer ~~is updated. Since, in~~ general, a file is either being read or written, most systems keep only one *current-file-position* pointer. Both the read and write operations use this same pointer, saving space and reducing the system complexity.

- **Repositioning within a file.** The directory is searched for the appropriate entry, and the current-file-position is set to a given value. Repositioning within a file does not need to involve any actual I/O. This file operation is also known as a file *seek*.
- **Deleting a file.** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space (so that it can be reused by other files) and erase the directory entry.
- **Truncating a file.** There are occasions when the user wants the attributes of a file to remain the same, but wants to erase the contents of the file. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged (except for file length) but for the file to be reset to length zero.

These six basic operations certainly comprise the minimal set of required file operations. Other common operations include *appending* new information to the end of an existing file, and *renaming* an existing file. These primitive operations may then be combined to perform other file operations. For instance, creating a *copy* of a file, or copying the file to another I/O device, such as a printer or a display, may be accomplished by creating a new file, and reading from the old and writing to the new. We also want to have operations that allow a user to get and set the various attributes of a file. For example, we may want to have an operation that allows a user to determine the status of a file, such as the file's length, and have an operation that allows a user to set file attributes, such as the file's owner.

Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems will *open* a file when that file first is used actively. The operating system keeps a small table containing information about all open files (the *open-file table*). When a file operation is requested, an index into this table is used, so no searching is required. When the file is no longer actively used, it is *closed* by the process and the operating system removes its entry in the open-file table.

Some systems implicitly open a file when the first reference is made to it. The file is automatically closed when the job or program that opened the file terminates. Most systems, however, require that a file be opened explicitly by the programmer with a system call (**open**) before that file can be used. The **open** operation takes a file name and searches the directory, copying the directory entry into the open-file table, assuming the file protections allow such access. The **open** system call will typically return a pointer to the entry in the open-file table. This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching, and simplifying the system-call interface.

The implementation of the **open** and **close** operations in a multiuser environment, such as UNIX, is more complicated. In such a system, several users may open the file at the same time. Typically, the operating system uses two levels of internal tables. There is a per-process table of all the files that each process has open. Stored in this table is information regarding the use of the file by the process. For instance, the current file pointer for each file is found here, indicating the location in the file which the next **read** or **write** call will affect.

Each entry in the per-process table in turn points to a systemwide open-file table. The systemwide table contains information that is process independent, such as the location of the file on disk, access dates, and file size. Once a file is opened by one process, another process executing an **open** call simply results in a new entry being added to the process' open file table with a new current file pointer, and a pointer to the appropriate entry in the systemwide table. Typically, the open-file table also has an *open count* associated with each file, indicating the number of processes which have the file open. Each **close** decreases this count, and when the count reaches zero, the file is no longer in use, and the file's entry is removed from the open file table. In summary, there are several pieces of information associated with an open file.

- **File pointer**. On systems that do not include a file offset as part of the **read** and **write** system calls, the system must track the last read/write location as a current-file-position pointer. This pointer is unique to each process operating on the file, and therefore must be kept separate from the on-disk file attributes.
- **File open count**. As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Because multiple processes may open a file, the system must wait for the last file close before removing the open-file table entry. This counter tracks the number of opens and closes, and reaches zero on the last close. The system can then remove the entry.
- **Disk location of the file**. Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory to avoid having to read it from disk for each operation.

Some operating systems provide facilities for locking sections of an open file for multiprocess access, to share sections of a file among several processes, and even to map sections of a file into memory on virtual-memory systems. This last function is called *memory mapping* a file and allows a part of the virtual address space to be logically associated with a section of a file. Reads and writes to that memory region are then treated as reads and writes to the file, greatly simplifying file use. Closing the file results in all the memory-mapped data being written back to disk and removed from the virtual memory of the process. Multiple processes may be allowed to map the same file into the virtual memory of each, to allow sharing of data. Writes by any of the processes modify the data in virtual memory and can be seen by all others that map the same section of the file. Given our knowledge of virtual memory from Chapter 9, it should be clear how the sharing of memory-mapped sections of memory is implemented: Each sharing process' virtual-memory map points to the same page of physical memory — the page that holds a copy of the disk block. This memory sharing is illustrated in Figure 10.1. So that access to the shared data is coordinated, the processes involved might use one of the mechanisms for achieving mutual exclusion described in Chapter 6.

10.1.3 File Types

One major consideration in designing a file system, and the entire operating system, is whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to print the binary-object form of a

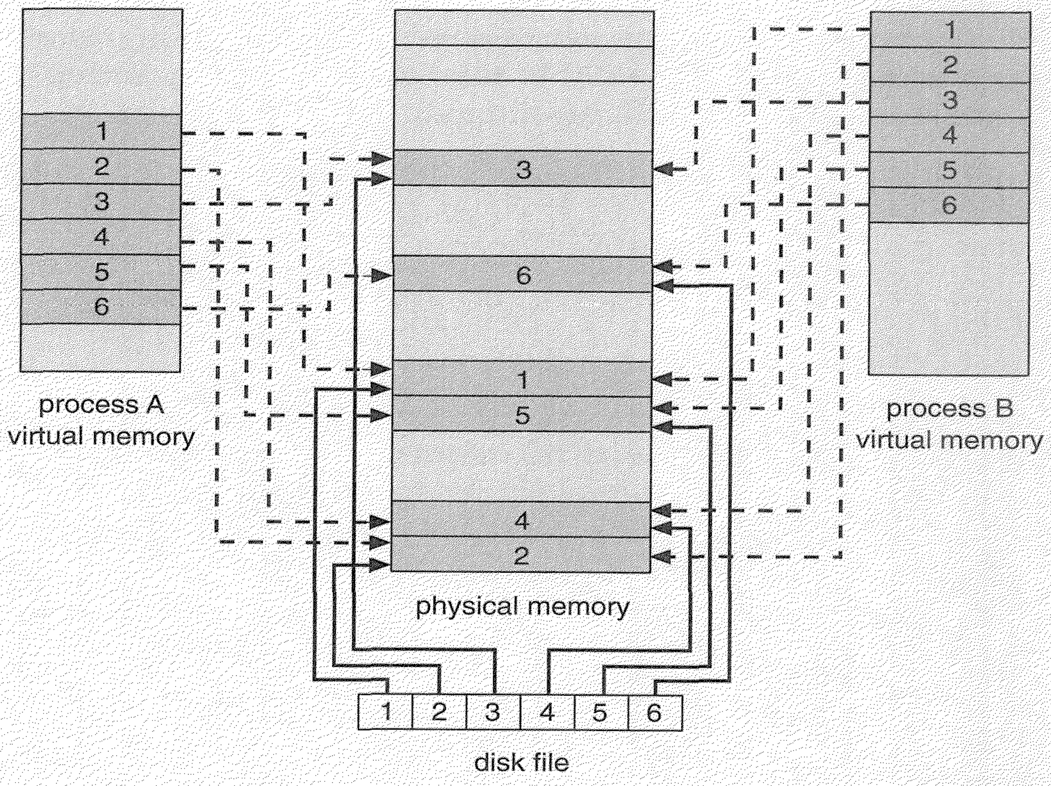


Figure 10.1 Memory-mapped files.

program. This attempt normally produces garbage, but can be prevented *if* the operating system has been told that the file is a binary-object program.

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts — a name and an *extension,* usually separated by a period character (Figure 10.2). In this way, the user and the operating system can tell from the name alone what the type of a file is. For example, in MS-DOS, a name can consist of up to eight characters followed by a period and terminated by an up-to-three-character extension. The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. For instance, only a file with a ".com", ".exe", or ".bat" extension can be executed. The ".com" and ".exe" files are two forms of binary executable files, whereas a ".bat" file is a *batch* file containing, in ASCII format,

File type	Usual extension	Function
Executable	exe, com, bin or none	ready-to-run machine-language program
Object	obj, o	compiled, machine language, not linked
Source code	c, p, pas, f77, asm, a	source code in various languages
Batch	bat, sh	commands to the command interpreter
Text	txt, doc	textual data, documents
Word processor	wp, tex, rrf, etc	various word-processor formats
Library	lib, a	libraries of routines for programmers
Print or view	ps, dvi, gif	ASCII or binary file in a format for printing or viewing
Archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage

Figure 10.2 Common file types.

commands to the operating system. MS-DOS recognizes only a few extensions, but application programs also use them to indicate file types in which they are interested. For example, assemblers expect source files to have an ".asm" extension, and the WordPerfect wordprocessor expects its file to end with ".wp". These extensions are not required, but a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered as "hints" to applications which operate on them.

Another example of the utility of file types comes from the TOPS-20 operating system. If the user tries to execute an object program whose source file has been modified (edited) since the object file was produced, the source file will be recompiled automatically. This function ensures that the user always runs an up-to-date object file. Otherwise, the user could waste a significant amount of time executing the old object file. Notice that, for this function to be possible, the operating system must be able to discriminate the source file from the object file, to check the time that each file was last modified or created, and to determine the language of the source program (in order to use the correct compiler).

Consider the Apple Macintosh operating system. In this system, each file has a type, such as "text" or "pict". Each file also has a creator attribute containing the name of the program that created it. This attribute is set by the operating system during the create call, so its use is enforced and supported by the system. For instance, a file produced by a word processor has the word processor's name as its creator. When the user opens that file, by double-clicking the mouse on the icon representing the file, the word processor is invoked automatically, and the file is loaded, ready to be edited.

The UNIX operating system is unable to provide such a feature because it uses a crude *magic number* stored at the beginning of some files to indicate roughly the type of the file: executable program, batch file (known as a shell script), postscript file, and so on. Not all files have magic numbers, so system features cannot be based solely on this type information. UNIX does not record the name of the creating program, either. UNIX also allows file name extension hints, but these extensions are not enforced or depended on by the operating system; they are mostly to aid users in determining the type of contents of the file.

10.1.4 File Structure

File types also may be used to indicate the internal structure of the file. As mentioned in Section 10.1.3, source and object files have structures that match the expectations of the programs that read them. Further, certain files must conform to a required structure that is understood by the operating system. For example, the operating system may require that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. Some operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures. For instance, DEC's popular VMS operating system has a file system that does support multiple file structures. It defines three file structures.

The above discussion brings us to one of the disadvantages of having the operating system support multiple file structures: The resulting size of the operating system is cumbersome. If the operating system defines five different file structures, it needs to contain the code to support these file structures. In addition, every file may need to be definable as one of the file types supported by the operating system. Severe problems may result from new applications that require information structured in ways not supported by the operating system.

For example, assume that a system supports two types of files: text files (composed of ASCII characters separated by a carriage return and line feed) and executable binary files. Now, if we (as users) want to define an encrypted file to protect our files from being read by unauthorized people, we may find neither file type to be appropriate. The encrypted file is not ASCII text lines, but rather is (apparently) random bits. Although it may appear to be a binary file, it is not executable. As a result, we may have to circumvent or misuse the operating system's file-types mechanism, or to modify or abandon our encryption scheme.

Some operating systems impose (and support) a minimal number of file structures. This approach has been adopted in UNIX, MS-DOS, and others. UNIX considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system. This scheme provides maximum flexibility, but little support. Each application program must include its own code to interpret an input file into the appropriate structure. However, all operating systems must support at least one structure — that of an executable file — so that the system is able to load and run programs.

Another example of an operating system that supports a minimal number of file structures is the Macintosh Operating System, which expects files to contain two parts: a *resource fork* and a *data fork.* The resource fork contains information of interest to the user. For instance, it holds the labels of any buttons displayed by the program. A foreign user may want to relabel these buttons in his own language, and the Macintosh operating system provides tools to allow modification of the data in the resource fork. The data fork contains the program code and data: the traditional file contents. To accomplish the same task on a UNIX or MS-DOS system, the programmer would need to change and recompile the source code, unless she created her own user-changeable data file. The moral of this example is that it is useful for an operating system to support structures that will be used frequently, and that will save the programmer substantial effort. Too few structures make programming inconvenient, whereas too many cause operating-system bloat and programmer confusion.

10.1.5 Internal File Structure

Internally, locating an offset within a file can be complicated for the operating system. Recall from Chapter 2 that disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. *Packing* a number of logical records into physical blocks is a common solution to this problem.

For example, the UNIX operating system defines all files to be simply a stream of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical record is 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks (say, 512 bytes per block) as necessary.

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user's application program or by the operating system.

In either case, the file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks. The conversion from logical records to physical blocks is a relatively simple software problem.

Notice that disk space being always allocated in blocks has the result that, in general, some portion of the last block of each file may be wasted. If each block is 512 bytes, then a file of 1949 bytes would be allocated four blocks (2048 bytes); the last 99 bytes would be wasted. The wasted bytes allocated to keep everything in units of blocks (instead of bytes) is *internal fragmentation*. All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

10.2 ■ Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. There are several ways that the information in the file can be accessed. Some systems provide only one access method for files. On other systems, such as those of IBM, many different access methods are supported, and choosing the right one for a particular application is a major design problem.

10.2.1 Sequential Access

The simplest access method is *sequential access*. Information in the file is processed in order, one record after the other. This mode of access is by far the most common, for example, editors and compilers usually access files in this fashion.

The bulk of the operations on a file are reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning, and, on some systems, a program may be able to skip forward or backward n records, for some integer n (perhaps only for $n = 1$). Sequential access is depicted in Figure 10.3. Sequential access is based on a tape model of a file, and works as well on sequential-access devices as it does on random-access ones.

10.2.2 Direct Access

Another method is *direct access* (or *relative access*). A file is made up of fixed-length *logical records* that allow programs to read and write records rapidly in no particular order. The direct access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows arbitrary blocks to be read or written. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer, and then read that block directly to provide the desired information.

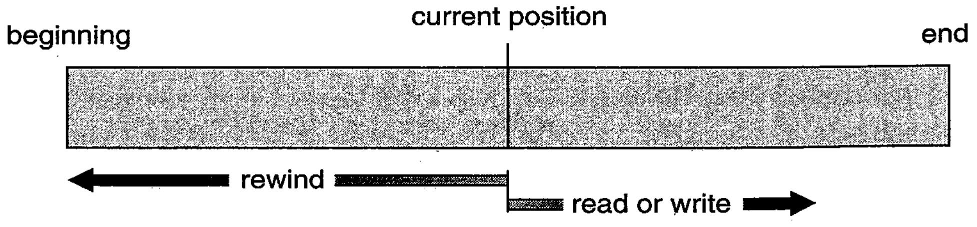


Figure 10.3 Sequential-access file.

For example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set, such as people, we might compute a hash function on the people's names, or search a small in-core index to determine a block to read and search.

The file operations must be modified to include the block number as a parameter. Thus, we have *read n,* where *n* is the block number, rather than *read next,* and *write n,* rather than *write next.* An alternative approach is to retain *read next* and *write next,* as with sequential access, and to add an operation, *position file to n,* where *n* is the block number. Then, to effect a *read n,* we would *position to n* and then *read next.*

The block number provided by the user to the operating system is normally a *relative block number.* A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block, and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed (called the allocation problem, as discussed in Chapter 11), and helps to prevent the user from accessing portions of the file system that may not be part of his file. Some systems start their relative block numbers at 0; others start at 1.

Given a logical record length *L*, a request for record *N* is turned into an I/O request for *L* bytes at location *L* + (*N* -1) within the file. Since logical records are of a fixed size, it is also easy to read, write, or delete a record.

Not all operating systems support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created; such a file can be accessed only in a manner consistent with its declaration. Notice, however, that it is easy to simulate sequential access on a direct-access file. If we simply keep a variable *cp,* which defines our current position, then we can simulate sequential file operations, as shown in Figure 10.4. On the other hand, it is extremely inefficient and clumsy to simulate a direct-access file on a sequential-access file.

10.2.3 Other Access Methods

Other access methods can be built on top of a direct-access method. These additional methods generally involve the construction of an *index* for the file. The index, like an index in the back of a book, contains pointers to the various blocks. To find an entry in the file, we first search the index, and then use the pointer to access the file directly and to find the desired entry.

Sequential access	Implementation for direct access
reset	*cp* := 0;

read next	*read cp;*
	cp := *cp*+1;

write next	*write cp;*
	cp := *cp*+1;

Figure 10.4 Simulation of sequential access on a direct-access file.

For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each entry consists of a 10-digit UPC and a six-digit price, for a 16-byte entry. If our disk has 1024 bytes per block, we can store 64 entries per block. A file of 120,000 entries would occupy about 2000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block. This index would have 2000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory. To find the price of a particular item, we can (binary) search the index. From this search, we would know exactly which block contains the desired entry and access that block. This structure allows us to search a large file doing little I/O.

With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files, which would point to the actual data items.

For example, IBM's indexed sequential access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. Figure 10.5 shows a similar situation as implemented by VMS index and relative files.

10.3 ■ Directory Structure

The file systems of computers can be extensive. Some systems store thousands of files on hundreds of gigabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two

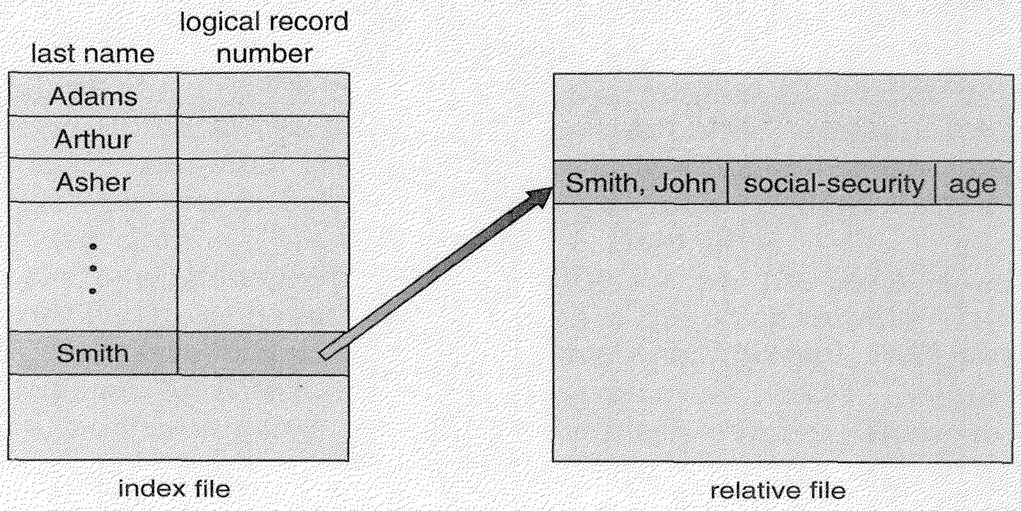


Figure 10.5 Example of index and relative files.

parts. First, the file system is broken into *partitions*, also known as *minidisks* in the IBM world or *volumes* in the PC and Macintosh arenas. Typically, each disk on a system contains at least one partition, which is a low-level structure in which files and directories reside. Sometimes, partitions are used to provide several separate areas within one disk, each treated as a separate storage device, whereas other systems allow partitions to be larger than a disk to group disks into one logical structure. In this way, the user needs to be concerned with only the logical directory and file structure, and can ignore completely the problems of physically allocating space for files. For this reason partitions can be thought of as virtual disks.

Second, each partition contains information about files within it. This information is kept in entries in a *device directory* or *volume table of contents*. The device directory (more commonly known simply as a "directory") records information — such as name, location, size, and type — for all files on that partition. Figure 10.6 shows the typical file-system organization.

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, then it becomes apparent that the directory itself can be organized in many ways. We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory. In Chapter 11, we discuss the appropriate data structures that can be used in the implementation of the directory structure. In this section, we examine several schemes for defining the logical structure of the directory system. When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

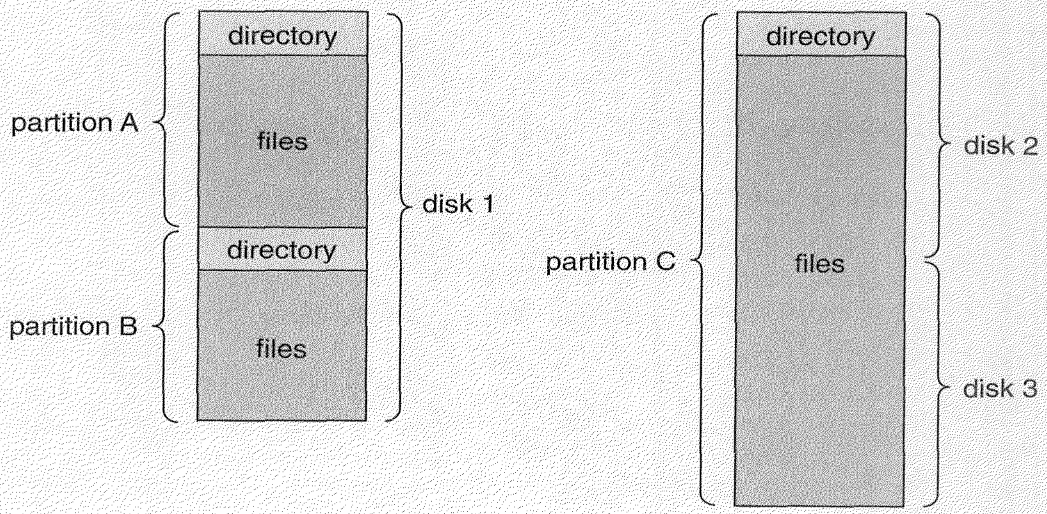


Figure 10.6 A typical file-system organization.

- **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.
- **Create a file**. New files need to be created and added to the directory.
- **Delete a file**. When a file is no longer needed, we want to remove it from the directory.
- **List a directory**. We need to be able to list the files in a directory, and the contents of the directory entry for each file in the list.
- **Rename a file**. Because the name of a file represents its contents to its users, the name must be changeable when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- **Traverse the file system**. It is useful to be able to access every directory, and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. This saving often consists of copying all files to magnetic tape. This technique provides a backup copy in case of system failure or if the file is simply no longer in use. In this case, the file can be copied to tape, and the disk space of that file released for reuse by another file.

In Sections 10.3.1 to 10.3.5, we describe the most common schemes for defining the logical structure of a directory.

10.3.1 Single-Level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 10.7).

A single-level directory has significant limitations, however, when the number of files increases or when there is more than one user. Since all files are in the same directory, they must have unique names. If we have two users who call their data file *test*, then the unique-name rule is violated. (For example, in one programming class, 23 students called the program for their second assignment *prog2*; another 11 called it *assign2*.) Although file names are generally selected to reflect the content of the file, they are often limited in length. The MS-DOS operating system allows only 11-character file names; UNIX allows 255 characters.

Even with a single user, as the number of files increases, it becomes difficult to remember the names of all the files, so as to create only files with unique names. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. In such an environment, keeping track of so many files is a daunting task.

10.3.2 Two-Level Directory

The major disadvantage to a single-level directory is the confusion of file names between different users. The standard solution is to create a *separate* directory for each user.

In the two-level directory structure, each user has her own user file directory (UFD). Each UFD has a similar structure, but lists only the files of a single user. When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The master file directory is indexed by user name or account number, and each entry points to the UFD for that user (Figure 10.8).

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.

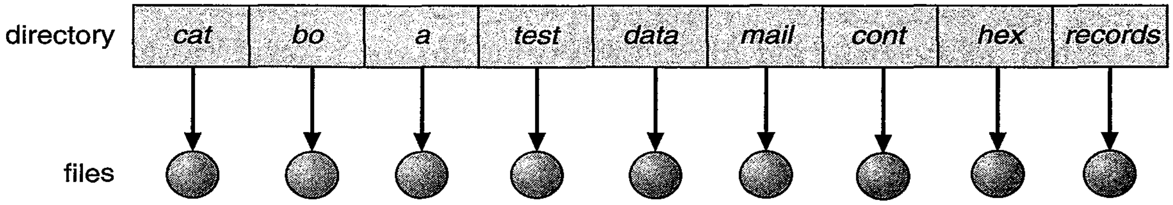


Figure 10.7 Single-level directory.

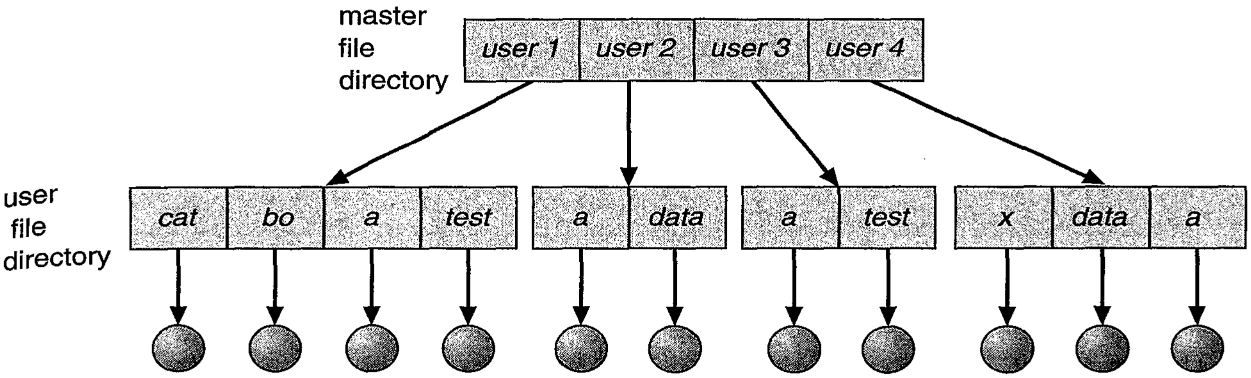


Figure 10.8 Two-level directory structure.

To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new user file directory and adds an entry for it to the master file directory. The execution of this program might be restricted to system administrators. The allocation of disk space for user directories can be handled with the techniques discussed in Chapter 11 for files themselves.

The two-level directory structure solves the name-collision problem, but it still has problems. This structure effectively isolates one user from another. This isolation is an advantage when the users are completely independent, but is a disadvantage when the users *want* to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users.

If access is to be permitted, one user must have the ability to name a file in another user's directory. To name a particular file uniquely in a two-level directory, we must give both the user name and the file name. A two-level directory can be thought of as a tree, or at least an inverted tree, of height 2. The root of the tree is the master file directory. Its direct descendants are the UFDs. The descendants of the user file directories are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the master file directory) to a leaf (the specified file). Thus, a user name and a file name define a *path name*. Every file in the system has a path name. To name a file uniquely, a user must know the path name of the file desired.

For example, if user A wishes to access her own test file named *test*, she can simply refer to *test*. To access the test file of user B (with directory-entry name *userb*), however, she might have to refer to */userb/test*. Every system has its own syntax for naming files in directories other than the user's own.

There is additional syntax to specify the partition of a file. For instance, in MS-DOS a partition is specified by a letter followed by a colon. Thus, a file specification might be "C:\userb\test". Some systems go even further and separate the partition, directory name, and file name parts of the specification. For instance, in VMS, the file "login.com" might be specified as: "u:[sst.jdeck]login.com;1", where "u" is the name of the partition, "sst" is the name of the directory, "jdeck" is the name of subdirectory, and "1", is the version number. Other systems simply treat the partition name as part of the directory name. The first name given is that of the partition, and the rest is the directory and file. For instance, "/u/pbg/test" might specify partition "u", directory "pbg", and file "test".

A special case of this situation occurs in regard to the system files. Those programs provided as a part of the system (loaders, assemblers, compilers, utility routines, libraries, and so on) are generally defined as files. When the appropriate commands are given to the operating system, these files are read by the loader and are executed. Many command interpreters act by simply treating the command as the name of a file to load and execute. As the directory system is defined presently, this file name would be searched for in the current user file directory. One solution would be to copy the system files into each user file directory. However, copying all the system files would be enormously wasteful of space. (If the system files require 5 megabytes, then supporting 12 users would require 5 × 12 = 60 megabytes just for copies of the system files.)

The standard solution is to complicate the search procedure slightly. A special user directory is defined to contain the system files (for example, user 0). Whenever a file name is given to be loaded, the operating system first searches the local user file directory. If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files. The sequence of directories searched when a file is named is called the *search path*. This idea can be extended, such that the search path contains an unlimited list of directories to search when a command name is given. This method is the one most used in UNIX and MS-DOS.

10.3.3 Tree-Structured Directories

Once we have seen how to view a two-level directory as a two-level tree, the natural generalization is to extend the directory structure to a tree of arbitrary height (Figure 10.9). This generalization allows users to create their own subdirectories and to organize their files accordingly. The MS-DOS system, for instance, is structured as a tree. In fact a tree is the most common directory structure. The tree has a root directory. Every file in the

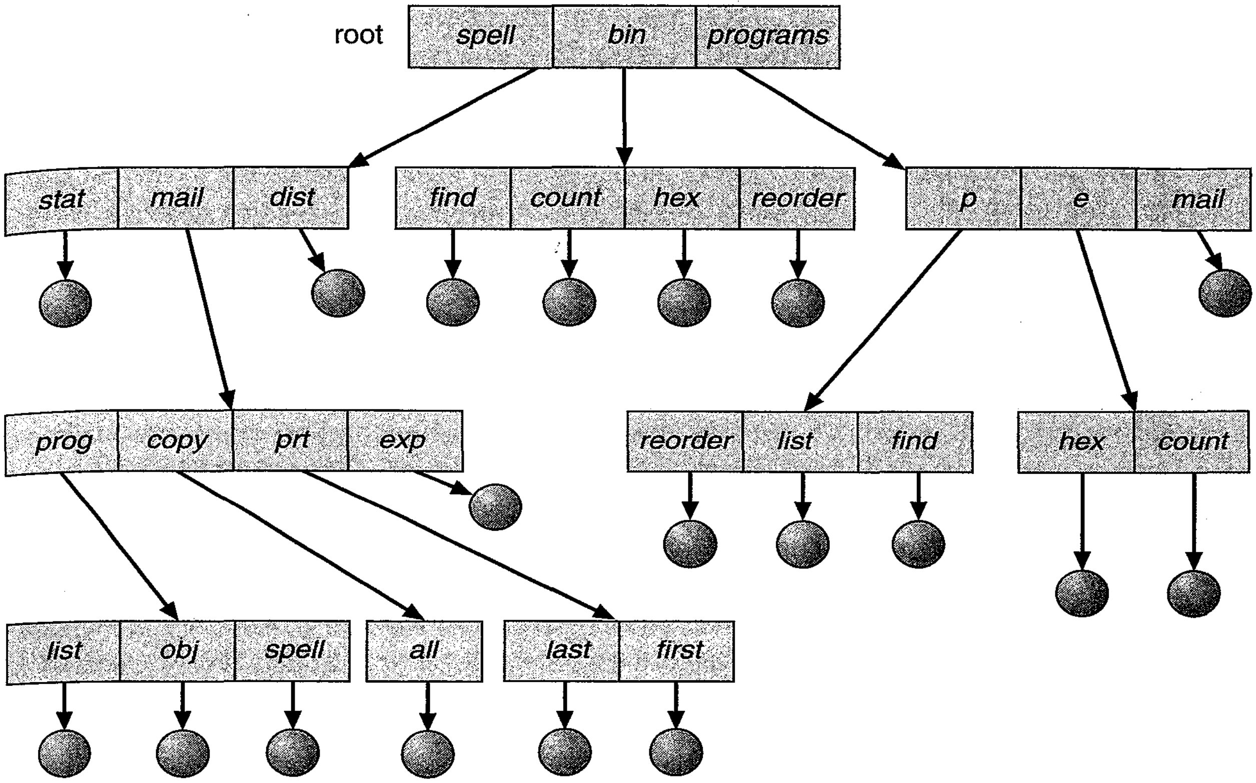


Figure 10.9 Tree-structured directory structure.

system has a unique path name. A path name is the path from the root, through all the subdirectories, to a specified file.

A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.

In normal use, each user has a *current directory*. The current directory should contain most of the files that are of current interest to the user. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user must either specify a path name or change the current directory to be the directory holding that file. To change the current directory to a different directory, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory. Thus, the user can change his current directory whenever he desires. From one **change directory** system call to the next, all **open** system calls search the current directory for the specified file.

The initial current directory of a user is designated when the user job starts or the user logs in. The operating system searches the accounting file (or some other predefined location) to find an entry for this user (for accounting purposes). In the accounting file is a pointer to (or the name of) the user's initial directory. This pointer is copied to a local variable for this user, which specifies the user's initial current directory.

Path names can be of two types: *absolute* path names or *relative* path names. An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path. A relative path name defines a path from the current directory. For example, in the tree-structured file system of Figure 10.9, if the current directory is *root/spell/mail*, then the relative path name *prt/first* refers to the same file as does the absolute path name *root/spell/mail/prt/first*.

Allowing the user to define his own subdirectories permits him to impose a structure on his files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book) or different forms of information (for example, the directory *programs* may contain source programs; the directory *bin* may store all the binaries).

An interesting policy decision in a tree-structured directory structure is how to handle the deletion of a directory. If a directory is empty, its entry in its containing directory can simply be deleted. However, suppose the directory to be deleted is not empty, but contains several files, or possibly subdirectories. One of two approaches can be taken. Some systems, such as MS-DOS, will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If there are any subdirectories, this procedure must be applied recursively to them, so that they can be deleted also. This approach may result in a substantial amount of work.

An alternative approach, such as that taken by the UNIX **rm** command, is to provide the option that, when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Note that either approach is fairly easy to implement; the choice is one of policy. The latter policy is more convenient, but more dangerous, because an entire directory structure may be removed with one command. If that command was issued in error, a large number of files and directories would need to be restored from backup tapes.

With a tree-structured directory system, files of other users can be accessed easily. For example, user B can access files of user A by specifying their path names. User B can specify either an absolute or a relative path name. Alternatively, user B could change her current directory to be user A's directory, and access the files by their file names. Some systems also allow users to define their own search paths. In this case, user B could define her search path to be (1) her local directory, (2) the system file directory, and (3) user A's directory, in that order. As long as the name of a file of user A did not conflict with the name of a local file or system file, it could be referred to simply by its name.

Note that a path to a file in a tree-structured directory can be longer than that in a two-level directory. To allow users to access programs without having to remember these long paths, the Macintosh operating system automates the search for executable programs. It maintains a file, called the "Desktop File", containing the name and location of all executable programs it has seen. When a new hard disk or floppy disk is added to the system, or the network is accessed, the operating system traverses the directory structure, searching for executable programs on the device and recording the pertinent information. This mechanism supports the double-click execution functionality described previously. A double-click on a file causes its creator attribute to be read, and the "Desktop File" to be searched for a match. Once the match is found, the appropriate executable program is started with the clicked-on file as its input.

10.3.4 Acyclic-Graph Directories

Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. The common subdirectory should be *shared*. A shared directory or file will exist in the file system in two (or more) places at once. Notice that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, there is only *one* actual file, so any changes made by one person would be immediately visible to the other. This form of sharing is particularly important for shared subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

A tree structure prohibits the sharing of files or directories. An *acyclic graph* allows directories to have shared subdirectories and files (Figure 10.10). The *same* file or subdirectory may be in two different directories. An acyclic graph (that is, a graph with no cycles) is a natural generalization of the tree-structured directory scheme.

In a situation where several people are working as a team, all the files to be shared may be put together into one directory. The user file directories of all the team members would each contain this directory of shared files as a subdirectory. Even when there is a single user, his file organization may require that some files be put into several different subdirectories. For example, a program written for a particular project should be both in the directory of all programs and in the directory for that project.

Shared files and subdirectories can be implemented in several ways. A common way, exemplified by many of the UNIX systems, is to create a new

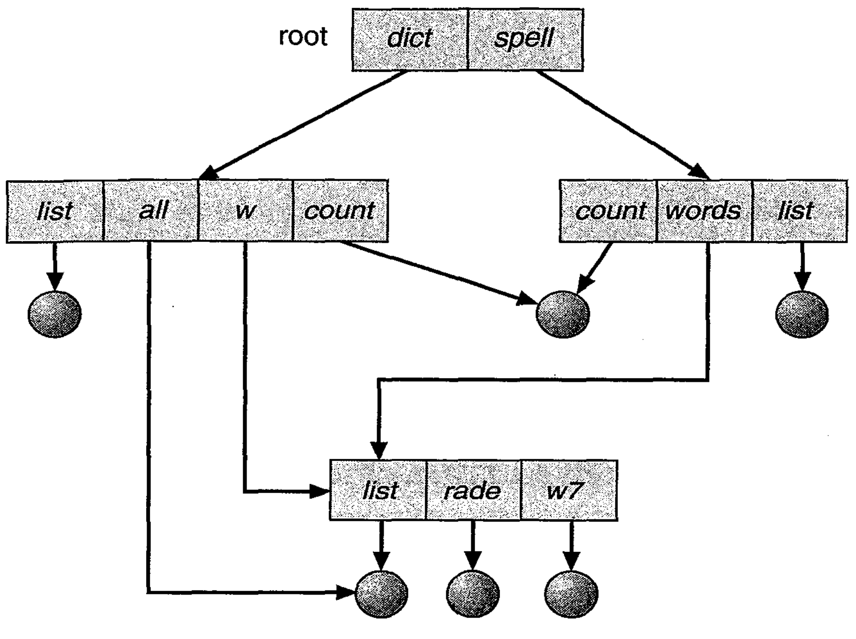


Figure 10.10 Acyclic-graph directory structure.

directory entry called a *link*. A link is effectively a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or relative path name (a *symbolic link*). When a reference to a file is made, we search the directory. The directory entry is marked as a link and the name of the real file (or directory) is given. We *resolve* the link by using the path name to locate the real file. Links are easily identified by their format in the directory entry, (or by their having a special type on systems that support types) and are effectively named indirect pointers. The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

The other approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal. A link is clearly different from the original directory entry; thus, the two are not equal. Duplicate directory entries, however, make the original and the copy indistinguishable. A major problem with duplicate directory entries is maintaining consistency if the file is modified.

An acyclic-graph directory structure is more flexible than is a simple tree structure, but is also more complex. Several problems must be considered carefully. Notice that a file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system (to find a file, to accumulate statistics on all files, or to copy all files to backup storage), this problem becomes significant, since we do not want to traverse shared structures more than once.

Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.

In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them also, but unless a list of the associated links is kept with each file, this search can be expensive. Alternatively, we can leave the links until an attempt is made to use them. At that time, we can determine that the file of the name given by the link does not exist, and can fail to resolve the link name; the access is treated just like any other illegal file name. (In this case, the system designer should consider carefully what to do when a file is deleted and another file of the same name is created, before a symbolic link to the original file is used.) In the case of UNIX, symbolic links are left when a file is deleted, and it is up to the user to realize that the original file is gone or has been replaced.

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted. We could keep a list of all references to a file (directory entries or symbolic links). When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

The trouble with this approach is the variable and potentially large size of the file-reference list. However, we really do not need to keep the entire list — we need to keep only a count of the *number* of references. A new link or directory entry increments the reference count; deleting a link or entry decrements the count. When the count is 0, the file can be deleted; there are no remaining references to it. The UNIX operating system uses this approach for nonsymbolic links, or *hard links*, keeping a reference count in the file information block (or inode, see Section 19.7.2). By effectively prohibiting multiple references to directories, we maintain an acyclic-graph structure.

To avoid these problems, some systems do not allow shared directories or links. For example, in MS-DOS, the directory structure is a tree structure, rather than an acyclic graph, thereby avoiding the problems associated with file deletion in an acyclic-graph directory structure.

10.3.5 General Graph Directory

One serious problem with using an acyclic graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure (Figure 10.11).

The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. We want to avoid traversing shared sections of an acyclic graph twice, mainly for performance reasons. If we have just searched a major shared subdirectory for a particular file, without finding that file, we want to avoid searching that subdirectory again; the second search would be a waste of time.

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating. One solution is to arbitrarily limit the number of directories which will be accessed during a search.

A similar problem exists when we are trying to determine when a file can be deleted. As with acyclic-graph directory structures, a 0 in the reference count means that there are no more references to the file or

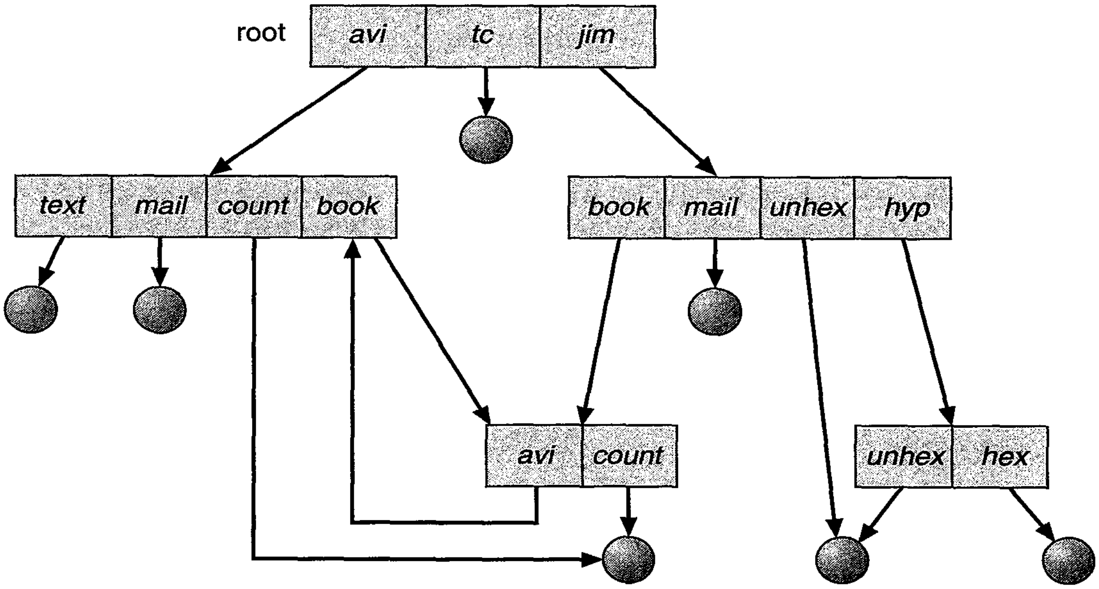


Figure 10.11 General graph directory.

directory, and the file can be deleted. However, it is also possible, when cycles exist, that the reference count may be nonzero, even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing (a cycle) in the directory structure. In this case, it is generally necessary to use *garbage collection* to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. (A similar marking procedure can be used to ensure that a traversal or search will cover everything in the file system once and only once.) Garbage collection for a disk-based file system, however, is extremely time-consuming and is thus seldom attempted.

Garbage collection is necessary only because of possible cycles in the graph. Thus, an acyclic-graph structure is much easier to work with. The difficulty is to avoid cycles, as new links are added to the structure. How do we know when a new link will complete a cycle? There are algorithms to detect cycles in graphs; however, they are computationally expensive, especially when the graph is on disk storage. Generally, tree directory structures are more common than are acyclic-graph structures.

10.4 ■ Protection

When information is kept in a computer system, a major concern is its protection from both physical damage (*reliability*) and improper access (*protection*).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost. Reliability is covered in more detail in Chapter 12.

Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multiuser system, however, other mechanisms are needed.

10.4.1 Types

The need for protecting files is a direct result of the ability to access files. On systems that do not permit access to the files of other users, protection is not needed. Thus, one extreme would be to provide complete protection by prohibiting access. The other extreme is to provide free access with no protection. Both of these approaches are too extreme for general use. What is needed is *controlled access*.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

- **Read**. Read from the file.
- **Write**. Write or rewrite the file.
- **Execute**. Load the file into memory and execute it.
- **Append**. Write new information at the end of the file.
- **Delete**. Delete the file and free its space for possible reuse.
- **List**. List the name and attributes of the file.

Other operations, such as renaming, copying, or editing the file, may also be controlled. For many systems, however, these higher-level functions (such as copying) may be implemented by a system program that makes lower-level system calls. Protection is provided at only the lower level. For instance, copying a file may be implemented simply by a sequence of read requests. In this case, a user with read access can also cause the file to be copied, printed, and so on.

Many different protection mechanisms have been proposed. Each scheme has its advantages and disadvantages and must be selected as appropriate for its intended application. A small computer system that is used by only a few members of a research group may not need the same types of protection as will a large corporate computer that is used for research, finance, and personnel operations. A complete treatment of the protection problem is deferred to Chapter 13.

10.4.2 Access Lists and Groups

The most common approach to the protection problem is to make access dependent on the identity of the user. Various users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an *access list*, specifying the user name and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

- Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
- The directory entry that previously was of fixed size needs now to be of variable size, resulting in space management being more complicated.

These problems can be resolved by use of a condensed version of the access list.

To condense the length of the access list, many systems recognize three classifications of users in connection with each file:

- **Owner**. The user who created the file is the owner.
- **Group**. A set of users who are sharing the file and need similar access is a group, or workgroup.
- **Universe**. All other users in the system constitute the universe.

As an example, consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named *book*. The protection associated with this file is as follows:

- Sara should be able to invoke all operations on the file.
- Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
- All others users should be able to read the file. (Sara is interested in letting as many people as possible read the text so that she can obtain appropriate feedback.)

To achieve such a protection, we must create a new group, say *text*, with members Jim, Dawn, and Jill. The name of the group *text* must be then associated with the file *book*, and the access right must be set in accordance with the policy we have outlined.

Note that, for this scheme to work properly, group membership must be controlled tightly. This control can be accomplished in a number of different ways. For example, in the UNIX system, groups can be created and modified by only the manager of the facility (or by any superuser). Thus, this control is achieved through human interaction. In the VMS system, with each file, an *access list* (also known as an *access control list*) may be associated, listing those users who can access the file. The owner of the file can create and modify this list. Access lists are discussed further in Section 13.4.2.

With this more limited protection classification, only three fields are needed to define protection. Each field is often a collection of bits, each of which either allows or prevents the access associated with it. For example, the UNIX system defines three fields of 3 bits each: **rwx**, where **r** controls read access, **w** controls write access, and **x** controls execution. A separate field is kept for the file owner, for the owner's group and for all other users. In this scheme, 9 bits per file are needed to record protection information. Thus, for our example, the protection fields for the file *book* are as follows; For the owner Sara, all 3 bits are set; for the group *text,* the **r** and **w** bits are set; and for the universe, only the **r** bit is set.

Notice, however, that this scheme is not as general as is the access-list scheme. To illustrate our point, let us return to the book example. Suppose that Sara wants to exclude Jason from the list of people who can read the text. She cannot do so with the basic protection scheme outlined.

10.4.3 Other Protection Approaches

There is another approach to the protection problem, which is to associate a password with each file. Just as access to the computer system itself is often controlled by a password, access to each file can be controlled by a password. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file to only those users who know the password. There are, however, several disadvantages to this scheme. First, if we associate a separate password with each file, the number of passwords that a user needs to remember may become large, making the scheme impractical. If only one password is used for all the files, then, once it is discovered, all files are accessible. Some systems (for example, TOPS-20) allow a user to associate a password with a subdirectory, rather than with an individual file, to deal with this problem. The IBM VM/CMS operating system allows three passwords for a minidisk: one each for read, write, and multiwrite access. Second, commonly, only one password is associated with each file. Thus, protection is on an all-or-nothing basis. To provide protection on a more detailed level, we must use multiple passwords.

Limited file protection is also currently available on single user systems, such as MS-DOS and Macintosh operating system. These operating systems, when originally designed, essentially ignored dealing with the protection problem. However, since these systems are being placed on networks where file sharing and communication is necessary, protection mechanisms are having to be retrofitted into the operating system. Note that it is almost always easier to design a feature into an new operating system than it is to add a feature to an existing one. Such updates are usually less effective and are not seamless.

```
-rw-rw-r--     1 pbg   staff      31200   Sep  3 08:30    intro.ps
drwx------     5 pbg   staff        512   Jul  8 09:33    private/
drwxrwxr-x     2 pbg   staff        512   Jul  8 09:35    doc/
drwxrwx---     2 pbg   student      512   Aug  3 14:13    student-proj/
-rw-r--r--     1 pbg   staff       9423   Feb 24 1993     program.c
-rwxr-xr-x     1 pbg   staff      20471   Feb 24 1993     program
drwx--x--x     4 pbg   faculty      512   Jul 31 10:31    lib/
drwx------     3 pbg   staff       1024   Aug 29 06:52    mail/
drwxrwxrwx     3 pbg   staff        512   Jul  8 09:35    test/
```

Figure 10.12 A sample directory listing.

We note that, in a multilevel directory structure, we need not only to protect individual files, but also to protect collections of files contained in a subdirectory; that is, we need to provide a mechanism for directory protection. The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory. In addition, we probably want to control whether a user can determine the existence of a file in a directory. Sometimes, knowledge of the existence and name of a file may be significant in itself. Thus, listing the contents of a directory must be a protected operation. Therefore, if a path name refers to a file in a directory, the user must be allowed access to both the directory and the file. In systems where files may have numerous path names (such as acyclic or general graphs), a given user may have different access rights to a file, depending on the path name used.

10.4.4 An Example: UNIX

In the UNIX system, directory protection is handled similarly to file protection. That is, associated with each subdirectory are three fields — owner, group, and universe — each consisting of the 3 bits **rwx**. Thus, a user can list the content of a subdirectory only if the **r** bit is set in the appropriate field. Similarly, a user can change his current directory to another current directory (say *foo*) only if the **x** bit associated with the *foo* subdirectory is set in the appropriate field.

A sample directory listing from a UNIX environment is shown in Figure 10.12. The first field describes the file or directory's protection. A **d** as the first character indicates a subdirectory. Also shown are the number of links to the file, the owner's name, the group's name, the size of the file in unit of bytes, the creation date, and finally the file's name (with optional extension).

10.5 ■ Consistency Semantics

Consistency semantics is an important criterion for evaluation of any file system that supports sharing of files. It is a characterization of the system that specifies the semantics of multiple users accessing a shared file simultaneously. In particular, these semantics should specify when modifications of data by one user are observable by other users.

For the following discussion, we assume that a series of file accesses (that is, reads and writes) attempted by a user to the same file, is always enclosed between the **open** and **close** operations. We call the series of accesses between an **open** and **close** operation a *file session.* To illustrate the concept, we sketch several prominent examples of consistency semantics.

10.5.1 UNIX Semantics

The UNIX file system (see Chapter 17) uses the following consistency semantics:

- Writes to an open file by a user are visible immediately to other users that have this file open at the same time.
- There is a mode of sharing where users share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.

These semantics lend themselves to an implementation where a file is associated with a single physical image that is accessed as an exclusive resource. Contention for this single image results in user processes being delayed.

10.5.2 Session Semantics

The Andrew file system (see Chapter 17) uses the following consistency semantics:

- Writes to an open file by a user are not visible immediately to other users that have the same file open simultaneously.
- Once a file is closed, the changes made to it are visible only in sessions starting later. Already-open instances of the file do not reflect these changes.

According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time. Consequently, multiple users are allowed to perform both read and write accesses concurrently on their image of the file, without delay. Notice that almost no constraints are enforced on scheduling accesses.

10.5.3 Immutable-Shared-Files Semantics

A different, unique approach is that of *immutable shared files*. Once a file is declared as shared by its creator, it cannot be modified. An immutable file has two important properties: Its name may not be reused and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed, rather than the file being a container for variable information. The implementation of these semantics in a distributed system (Chapter 17) is simple, since the sharing is disciplined (read-only).

10.6 ■ Summary

A file is an abstract data type defined and implemented by the operating system. It is a sequence of logical records. A logical record may be a byte, a line (fixed or variable length), or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.

The major task for the operating system is to map the logical file concept onto physical storage devices such as magnetic tape or disk. Since the physical record size of the device may not be the same as the logical record size, it may be necessary to block logical records into physical records. Again, this task may be supported by the operating system or left for the application program.

Tape-based file systems are constrained; most file systems are disk-based. Tapes are commonly used for data transport between machines, or for backup or archival storage.

Each device in a file system keeps a volume table of contents or device directory listing the location of the files on the device. In addition, it is useful to create directories to allow files to be organized. A single-level directory in a multiuser system causes naming problems, since each file must have a unique name. A two-level directory solves this problem by creating a separate directory for each user. Each user has her own directory, containing her own files.

The directory lists the files by name, and includes such information as the file's location on the disk, length, type, owner, time of creation, time of last use, and so on.

The natural generalization of a two-level directory is a tree-structured directory. A tree-structured directory allows a user to create subdirectories to organize his files. Acyclic-graph directory structures allow subdirectories and files to be shared, but complicate searching and deletion. A general graph structure allows complete flexibility in the sharing of files and directories, but sometimes requires garbage collection to recover unused disk space.

Since files are the main information-storage mechanism in most computer systems, file protection is needed. Access to files can be controlled separately for each type of access: read, write, execute, append, list directory, and so on. File protection can be provided by passwords, by access lists, or by special ad hoc techniques.

File systems are often implemented in a layered or modular structure. The lower levels deal with the physical properties of storage devices. Upper levels deal with symbolic file names and logical properties of files. Intermediate levels map the logical file concepts into physical device properties.

■ Exercises

10.1 Consider a file system where a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?

10.2 Some systems automatically delete all user files when a user logs off or a job terminates, unless the user explicitly requests that they be kept; other systems keep all files unless the user explicitly deletes them. Discuss the relative merits of each approach.

10.3 Why do some systems keep track of the type of a file, while others leave it to the user or simply do not implement multiple file types? Which system is "better?"

10.4 Similarly, some systems support many types of structures for a file's data, while others simply support a stream of bytes. What are the advantages and disadvantages?

10.5 What are the advantages and disadvantages of recording the name of the creating program with the file's attributes (as is done in the Macintosh operating system)?

10.6 Could you simulate a multilevel directory structure with a single-level directory structure in which arbitrarily long names can be used? If your answer is yes, explain how you can do so, and contrast this scheme with the multilevel directory scheme. If your answer is no, explain what prevents your simulation's success. How would your answer change if file names were limited to seven characters?

10.7 Explain the purpose of the **open** and **close** operations.

10.8 Some systems automatically open a file when it is referenced for the first time, and close the file when the job terminates. Discuss the advantages and disadvantages of this scheme as compared to the more traditional one, where the user has to open and close the file explicitly.

10.9 Give an example of an application in which data in a file should be accessed in the following order:

a. Sequentially

b. Randomly

10.10 Some systems provide file sharing by maintaining a single copy of a file; other systems maintain several copies, one for each of the users sharing the file. Discuss the relative merits of each approach.

10.11 In some systems, a subdirectory can be read and written by an authorized user, just as ordinary files can be.

a. Describe the protection problems that could arise.

b. Suggest a scheme for dealing with each of the protection problems you named in part a.

10.12 Consider a system that supports 5000 users. Suppose that you want to allow 4990 of these users to be able to access one file.

a. How would you specify this protection scheme in UNIX?

b. Could you suggest another protection scheme that can be used more effectively for this purpose than the scheme provided by UNIX?

10.13 Researchers have suggested that, instead of having an access list associated with each file (specifying which users can access the file, and how), we should have a *user control list* associated with each user (specifying which files a user can access, and how). Discuss the relative merits of these two schemes.

Bibliographic Notes

General discussions concerning file systems were offered by Grosshans [1986]. Golden and Pechura [1986] described the structure of microcomputer file systems. Database systems and their file structures were described in full in Korth and Silberschatz [1991].

A multilevel directory structure was first implemented on the MULTICS system [Organick 1972]. Most operating systems now implement multilevel directory structures. These include UNIX [Ritchie and Thompson 1974], the Apple Macintosh operating system [Apple 1991], and MS-DOS [Microsoft 1991].

The MS-DOS file system was described in Norton and Wilton [1988]. That of VAX VMS was covered in Kenah el al. [1988], and Digital [1981]. The Network File System (NFS) was designed by Sun Microsystems, and allows directory structures to be spread across networked computer systems. Discussions concerning NFS were presented in Sandberg et al. [1985], Sandberg [1987], and Sun Microsystems [1990]. NFS is fully described in Chapter 17. The immutable-shared-files semantics was described by Schroeder et al. [1985].

Interesting research is ongoing in the area of file-system interfaces. Several papers in USENIX [1992a] discussed the issues of file naming and attributes. For example, the Plan 9 operating system from AT&T Bell Laboratories makes all objects look like file systems. Thus, to display a list of processes on a system, a user simply lists the contents of the */proc* directory. Similarly, to display the time of day, a user needs only to type the file */dev/time*.