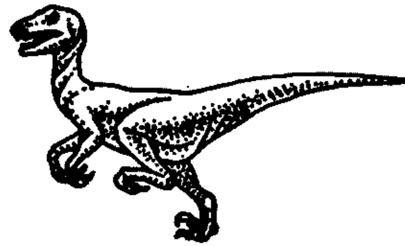
CHAPTER



In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**. We 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 the 20th 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 prevent or deal with deadlocks. Most current operating systems do not provide deadlock-prevention facilities, but such features will probably be added soon. Deadlock problems can only become more common, given current trends, including larger numbers of processes, multithreaded programs, many more resources within a system, and an 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 consisting of some number of identical instances. Memory space, CPU cycles, files, and I/O devices (such as printers and DVD 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 has only 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, if 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 2. Examples are the `request()` and `release()` device, `open()` and `close()` file, and `allocate()` and `free()` memory system calls. Request and release of resources that are not managed by the operating system can be accomplished through the `wait()` and `signal()` operations on semaphores or through acquisition and release of a mutex lock. For each use of a kernel-managed resource by a process or thread, the operating system checks to make sure that the process has requested and has been allocated the resource. A system table records whether each resource is free or allocated; for each resource that is allocated, the table also records the process to which it is allocated. 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 only by 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 facilities discussed in Chapter 3).

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

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

A programmer who is developing multithreaded applications must pay particular attention to this problem. Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Figure 7.1 Deadlock example.

Let's see how deadlock can occur in a multithreaded Pthread program using mutex locks. The `pthread_mutex_init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`.

Two mutex locks are created in the following code example:

```
/* Create and initialize the mutex locks */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

Next, two threads—`thread_one` and `thread_two`—are created, and both these threads have access to both mutex locks. `thread_one` and `thread_two` run in the functions `do_work_one()` and `do_work_two()`, respectively, as shown in Figure 7.1.

In this example, `thread_one` attempts to acquire the mutex locks in the order (1) `first_mutex`, (2) `second_mutex`, while `thread_two` attempts to acquire the mutex locks in the order (1) `second_mutex`, (2) `first_mutex`. Deadlock is possible if `thread_one` acquires `first_mutex` while `thread_two` aacquites `second_mutex`.

Note that, even though deadlock is possible, it will not occur if `thread_one` is able to acquire and release the mutex locks for `first_mutex` and `second_mutex` before `thread_two` attempts to acquire the locks. This example illustrates a problem with handling deadlocks: It is difficult to identify and test for deadlocks that may occur only under certain circumstances.

7.2

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at 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**. A process must be holding at least one resource and 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**. A set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ..., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource 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 different types

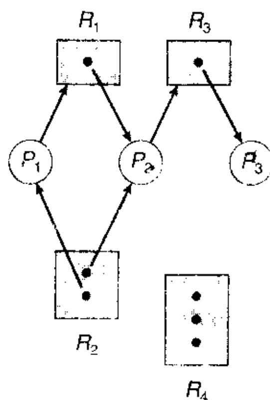


Figure 7.2 Resource-allocation graph.

of nodes: $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the active 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 has 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 rectangle. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle R_j, whereas an assignment edge must also designate one of the dots in the rectangle.

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 no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure 7.2 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 an instance of R_2 and is waiting for an instance of R_3.
  - Process P_3 is holding an instance of R_3.

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain 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 has 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, we return to the resource-allocation graph depicted in Figure 7.2. 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 7.3). At this point, two minimal cycles exist in the system:

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

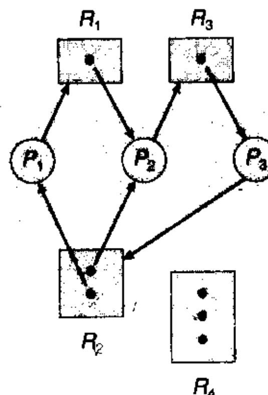


Figure 7.3 Resource-allocation graph with a deadlock.

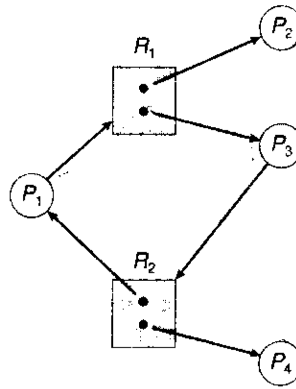


Figure 7.4 Resource-allocation graph with a cycle but no deadlock.

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 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 7.4. 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 deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

7.3

Generally speaking, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.

Next, we elaborate briefly on each of the three methods for handling deadlocks. Then, in Sections 7.4 through 7.7, we present detailed algorithms. However, before proceeding, we should mention that some researchers have

argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in operating systems. The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.

To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock-avoidance scheme. **Deadlock prevention** provides 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 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, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. 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 arise. 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 neither ensures that a deadlock will never occur nor provides a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlocked state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in deterioration of the system's performance, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems, as mentioned earlier. In many systems, deadlocks occur infrequently (say, once per year); thus, this method is cheaper than the prevention, avoidance, or detection and recovery methods, which must be used constantly. Also, in some circumstances, a system is in a frozen state but not in a deadlocked state. We see this situation, for example, with a real-time process running at the highest priority (or any process running on a nonpreemptive scheduler) and never returning control to the operating system. The system must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.

7.4

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. We 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, in contrast, 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, we cannot prevent deadlocks by denying the mutual-exclusion condition, because 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 it 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 DVD drive to a file on disk, sorts the 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 DVD drive, disk file, and 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 DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD 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.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. In the example given, for instance, we can release the DVD 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 for deadlocks 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 is holding some resources and requests another resource that cannot be immediately allocated

to it (that is, the process must wait), then all resources currently being held are preempted. In other words, 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, 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 neither available nor 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

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this 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.

To illustrate, we let $R = \{R_1, R_2, ..., R_m\}$ 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 $F: R \rightarrow N$, 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 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.

We can accomplish this scheme in an application program by developing an ordering among all synchronization objects in the system. All requests for synchronization objects must be made in increasing order. For example, if the lock ordering in the Pthread program shown in Figure 7.1 was

F(`first_mutex`) = 1
F(`second_mutex`) = 5

then `thread_two` could not request the locks out of order.

Keep in mind that developing an ordering, or hierarchy, in itself does not prevent deadlock. It is up to application developers to write programs that follow the ordering. Also note that the function F should be defined according to the normal order of usage of the resources in a system. For example, because the tape drive is usually needed before the printer, it would be reasonable to define F(tape drive) < F(printer).

Although ensuring that resources are acquired in the proper order is the responsibility of application developers, certain software can be used to verify that locks are acquired in the proper order and to give appropriate warnings when locks are acquired out of order and deadlock is possible. One lock-order verifier, which works on BSD versions of UNIX such as FreeBSD, is known as **witness**. Witness uses mutual-exclusion locks to protect critical sections, as described in Chapter 6; it works by dynamically maintaining the relationship of lock orders in a system. Let's use the program shown in Figure 7.1 as an example. Assume that `thread_one` is the first to acquire the locks and does so in the order (1) `first_mutex`, (2) `second_mutex`. Witness records the relationship that `first_mutex` must be acquired before `second_mutex`. If `thread_two` later acquires the locks out of order, witness generates a warning message on the system console.

7.5

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, the system might need to know that process P will request first the tape drive and then the printer before releasing both resources, whereas process Q will request first the printer and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or

not the process should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

The various algorithms that use this approach 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 this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. Such an algorithm defines the deadlock-avoidance approach. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation *state* is defined by the number of available and allocated resources and the maximum demands of the processes. In the following sections, we explore two deadlock-avoidance algorithms.

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 $<P_1, P_2, ..., P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j, with $j < i$. In this situation, if the resources that 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 deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 7.5). An unsafe state *may* lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) 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.

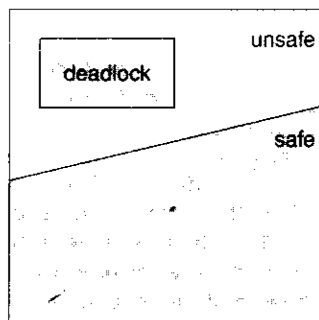


Figure 7.5 Safe, unsafe, and deadlock state spaces.

To illustrate, we consider a system with 12 magnetic tape drives and three processes: P_0, P_1, and P_2. Process P_0 requires 10 tape drives, process P_1 may need as many as 4 tape drives, 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 tape drives, 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. 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 can get all its tape drives and return them (the system will then have all 12 tape drives available).

A system can go from a safe state to an unsafe state. Suppose that, at time t_1, process P_2 requests and is allocated one 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 will have only 4 available tape drives. Since process P_0 is allocated 5 tape drives but has a maximum of 10, it may 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 was 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.

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.

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 otherwise be.

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 already described, we introduce a new type of edge, called a **claim edge**. A claim edge $P_i \dashrightarrow R_j$ 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 in the graph by a dashed line. When process P_i requests resource

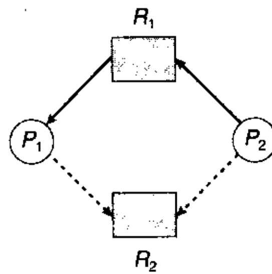


Figure 7.6 Resource-allocation graph for deadlock avoidance.

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 7.6. 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 7.7). 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.

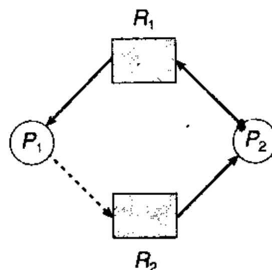


Figure 7.7 An unsafe state in a resource-allocation graph.

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 chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could 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] equals 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] equals 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] equals 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] equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that *Need*[i][j] equals *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, we next 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 $i = 1, 2, ..., n$. 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 $Need_i$. 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

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, ..., n - 1$.

Find an i such that both

a. $Finish[i] == false$

b. $Need_i \leq Work$

If no such i exists, go to step 4.

$Work = Work + Allocation_i$
$Finish[i] = true$
Go to step 2.

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 determine whether a state is safe.

7.5.3.2 Resource-Request Algorithm

We now describe the algorithm which determines if requests can be safely granted.

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:

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

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

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

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

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 $Request_i$, and the old resource-allocation state is restored.

7.5.3.3 An Illustrative Example

Finally, to illustrate the use of the banker's algorithm, consider a system with five processes P_0 through P_4 and three resource types A, B, and 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 as follows:

	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 $Request_1 = (1,0,2)$. To decide whether this request can be immediately granted, we first check that $Request_1 \leq Available$—that is, that $(1,0,2) \leq (3,3,2)$, 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 the 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. Furthermore, a request for (0,2,0) by P_0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

We leave it as a programming exercise to implement the banker's algorithm.

7.6

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, we 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 resource nodes 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

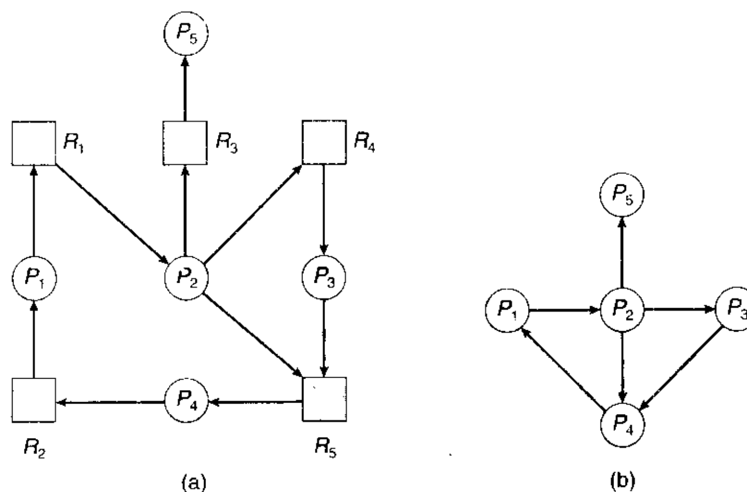


Figure 7.8 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

R_q. For example, in Figure 7.8, 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 *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. We turn now to a deadlock-detection algorithm that is applicable 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.

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 $Request[i][j]$ equals k, then process P_i is requesting k more instances of resource type R_j.

The $\leq$ relation between two vectors is defined as in Section 7.5.3. To simplify notation, we again treat the rows in the matrices *Allocation* and *Request* as vectors; we refer to them as $Allocation_i$ and $Request_i$. 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.

Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$. For $i = 0, 1, ..., n\text{-}1$, if $Allocation_i \neq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$.

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.

$Work = Work + Allocation_i$
$Finish[i] = true$
Go to step 2.

If $Finish[i] == false$, for some i, $0 \leq i < n$, then the system is in a deadlocked 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 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, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six 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 $<P_0, P_2, P_3, P_1, P_4>$ results 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:

- How *often* is a deadlock likely to occur?
- 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 occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes. In the extreme, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the deadlocked set of processes 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 create many cycles in the resource graph, each cycle completed by the most recent request and "caused" by the one identifiable process.

Of course, if the deadlock-detection algorithm is invoked for every resource request, this will 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 causes CPU utilization to drop.) If the detection algorithm is invoked at arbitrary points in time, there may be many cycles in the resource graph. In this case, 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 are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system *recover* from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other 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 great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to 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.

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

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may affect 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 deadlocked process is holding and the amount of time the 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. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all 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 must 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

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. There are three principal methods for dealing with deadlocks:

- Use some protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state, detect it, and then recover.
- Ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including UNIX and Windows.

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

A method for avoiding deadlocks that is less stringent than the prevention algorithms requires that the operating system have a priori information on how each process will utilize system resources. The banker's algorithm, for example, requires a priori information about 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.

Where preemption is used to deal with deadlocks, three issues must be addressed: selecting a victim, rollback, and starvation. In a system that selects victims for rollback primarily on the basis of cost factors, starvation may occur, and the selected process can never complete its designated task.

Finally, researchers have argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in operating systems. The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.

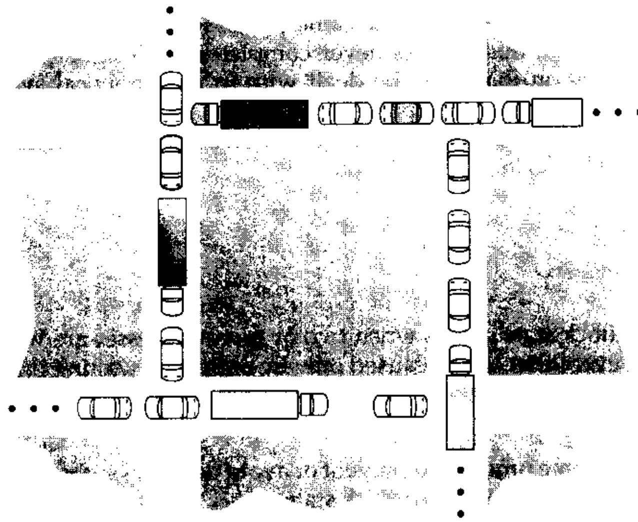


Figure 7.9 Traffic deadlock for Exercise 7.1.

Exercises

7.1 Consider the traffic deadlock depicted in Figure 7.9.

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

b. State a simple rule for avoiding deadlocks in this system.

7.2 Consider the deadlock situation that could occur in the dining-philosophers problem when the philosophers obtain the chopsticks one at a time. Discuss how the four necessary conditions for deadlock indeed hold in this setting. Discuss how deadlocks could be avoided by eliminating any one of the four conditions.

7.3 Compare the circular-wait scheme with the various deadlock-avoidance schemes (like the banker's algorithm) with respect to the following issues:

a. Runtime overheads

b. System throughput

7.4 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.

7.5 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 $m + n$.

7.6 Consider the dining-philosophers problem where the chopsticks are placed at the center of the table and any two of them could be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for determining whether a particular request could be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

7.7 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.

7.8 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?

7.9 Write a multithreaded program that implements the banker's algorithm discussed in Section 7.5.3. Create n threads that request and release resources from the bank. The banker will grant the request only if it leaves the system in a safe state. You may write this program using either Pthreads or Win32 threads. It is important that access to shared data is safe from concurrent access. Such data can be safely accessed using mutex locks, which are available in both the Pthreads and Win32 API. Coverage of mutex locks in both of these libraries is described in "producer–consumer problem" project in Chapter 6.

7.10 A single-lane bridge connects the two Vermont villages of North Tunbridge and South Tunbridge. Farmers in the two villages use this bridge to deliver their produce to the neighboring town. The bridge can become deadlocked if both a northbound and a southbound farmer get on the bridge at the same time (Vermont farmers are stubborn and are unable to back up.) Using semaphores, design an algorithm that prevents deadlock. Initially, do not be concerned about starvation (the situation in which northbound farmers prevent southbound farmers from using the bridge, or vice versa).

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. Starvation was covered by Holt [1972]. Hyman [1985] provided the deadlock example from the Kansas legislature. A recent study of deadlock handling is provided in Levine [2003].

The various prevention algorithms were suggested by Havender [1968], who 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]. Exercise 7.5 is from Holt [1971].

The deadlock-detection algorithm for multiple instances of a resource type, which was described in Section 7.6.2, was presented by Coffman et al. [1971].

Bach [1987] describes how many of the algorithms in the traditional UNIX kernel handle deadlock. Solutions to deadlock problems in networks is discussed in works such as Culler et al. [1998] and Rodeheffer and Schroeder [1991].

The witness lock-order verifier is presented in Baldwin [2002].

Part Four

Memory 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 the CPU and the speed of its response to users, the computer must keep several processes in memory. Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm depends on the situation. Selection of a memory-management scheme for a system depends on many factors, especially on the *hardware* design of the system. Each algorithm requires its own hardware support.

Memory-Management Strategies

CHAPTER

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; that is, 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 method 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, although recent designs have closely integrated the hardware and operating system.

8.1 Background

As we saw in Chapter 1, memory is central to the operation of a modern computer system. Memory consists of 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, first fetches 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. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore *how* a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

We begin our discussion by covering several issues that are pertinent to the various techniques for managing memory. This includes an overview of basic hardware issues, the binding of symbolic memory addresses to actual physical addresses, and distinguishing between logical and physical addresses.

We conclude with a discussion of dynamically loading and linking code and shared libraries.

8.1.1 Basic Hardware

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Memory access may take many cycles of the CPU clock to complete, in which case the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory. A memory buffer used to accommodate a speed differential, called a **cache**, is described in Section 1.8.3.

Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation has to protect the operating system from access by user processes and, in addition, to protect user processes from one another. This protection must be provided by the hardware. It can be implemented in several ways, as we shall see throughout the chapter. In this section, we outline one possible implementation.

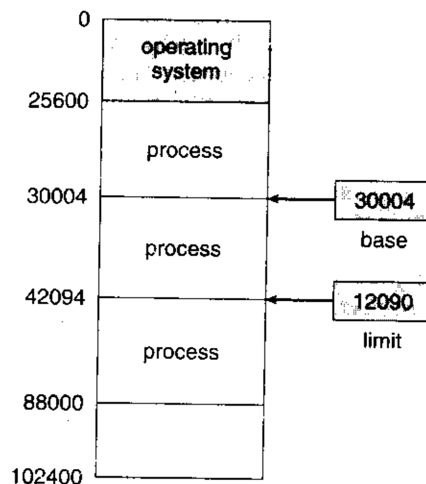


Figure 8.1 A base and a limit register define a logical address space.

We first need to make sure that each process has a separate memory space. To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 8.1. The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).

Protection of memory space is accomplished by having the CPU hardware compare *every* address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure 8.2). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

The operating system, executing in kernel mode, is given unrestricted access to both operating system and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, and so on.

8.1.2 Address Binding

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved

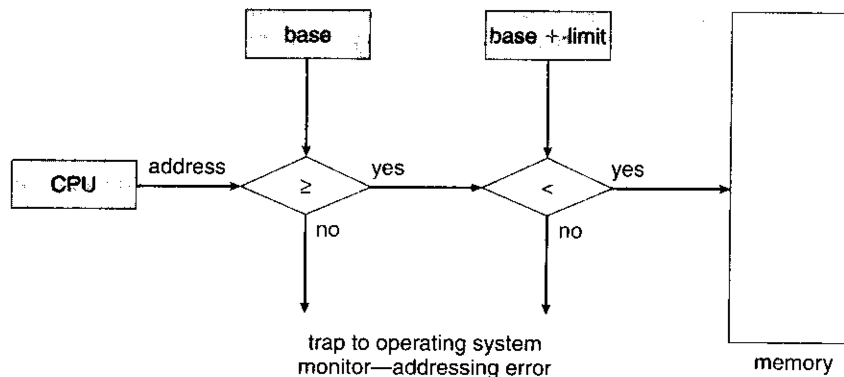


Figure 8.2 Hardware address protection with base and limit registers.

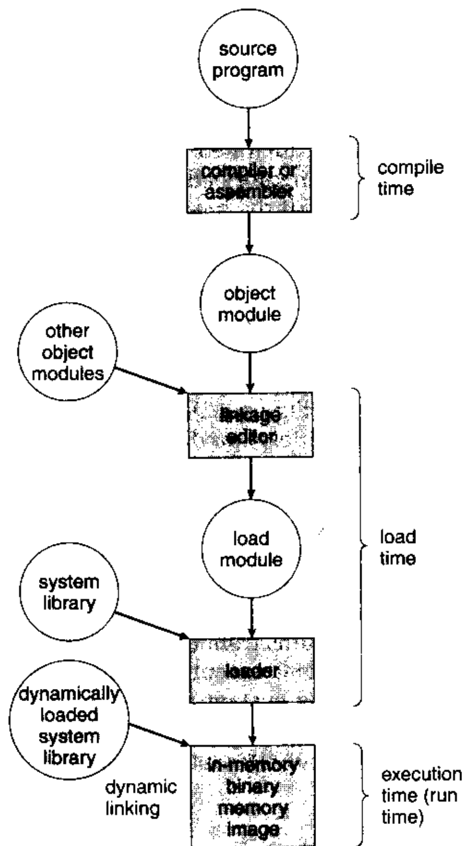


Figure 8.3 Multistep processing of a user program.

between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form 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 need not be 00000. This approach 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 8.3). 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 the 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 you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside 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 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 reload the user code to incorporate this changed value.
- **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 8.1.3. Most general-purpose operating systems use this method.

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

8.1.3 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 methods generate identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. 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 a **logical address space**; the set of all physical addresses corresponding to these logical addresses is 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 a hardware device called the **memory-management unit (MMU)**. We can choose from many different methods to accomplish such mapping, as we discuss in Sections 8.3 through 8.7. For the time being, we illustrate this mapping with a simple MMU scheme, which is a generalization of the base-register scheme described in Section 8.1.1. The base register is now called a **relocation register** The value in the relocation register is *added* to every address generated by user process at the time it is sent to memory (see Figure 8.4). For exam if the base is at 14000, then an attempt by the user to address locatic dynamically relocated to location 14000; an access to location 346 is

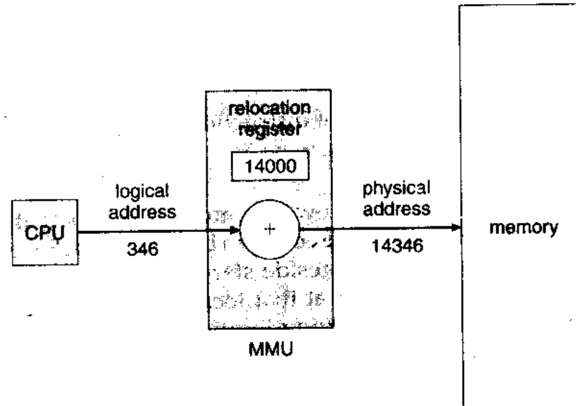


Figure 8.4 Dynamic relocation using a relocation register.

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.

The user program never sees the *real* physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with 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 8.1.2. The final location of a referenced memory address is not determined until the reference is made.

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.

8.1.4 Dynamic Loading

In our discussion so far, the entire program and all data of a process must be in physical memory for the process to execute. The size of a process is thus limited to the size of physical memory. 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 ··~e has been loaded. If not, the relocatable linking loader is called to load ··tine into memory and to update the program's address tables nge. Then control is passed to the newly loaded routine.

ıge of dynamic loading is that an unused routine is never ethod is particularly useful when large amounts of code are

/ a ·ple, ın 0 is mapped

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 used (and hence 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 method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

8.1.5 Dynamic Linking and Shared Libraries

Figure 8.3 also shows **dynamically linked libraries**. Some 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. Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) 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. The 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 the stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine and executes the routine. Thus, the next time that particular 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 help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses. We elaborate on this concept when we discuss paging in Section 8.4.4.

8.2 Swapping

A process must 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 another process into the memory space that has been freed (Figure 8.5). 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 some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. In addition, the quantum must be large enough to allow reasonable amounts of computing to be 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 and then 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 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 easily moved to a different location. If execution-time binding is being used, however, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

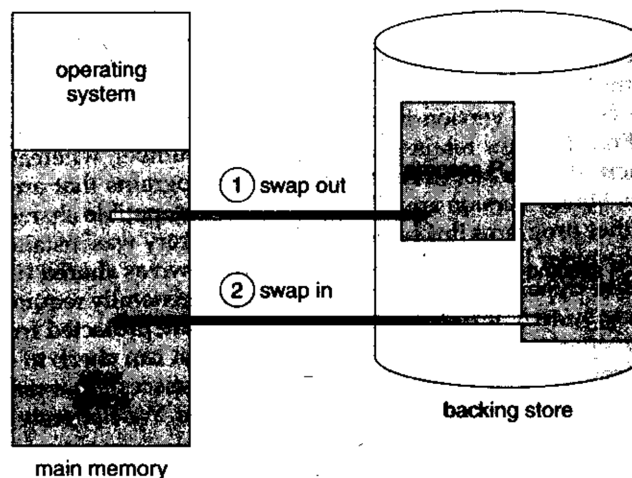


Figure 8.5 Swapping of two processes using a disk as a backing store.

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 it 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 it is not, and if 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 and transfers control to the selected process.

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 10 MB in size and the backing store is a standard hard disk with a transfer rate of 40 MB per second. The actual transfer of the 10-MB process to or from main memory takes

$$\begin{aligned} 10000 \text{ KB}/40000 \text{ KB per second} &= 1/4 \text{ second} \\ &= 250 \text{ milliseconds.} \end{aligned}$$

Assuming that no head seeks are necessary, and assuming an average latency of 8 milliseconds, the swap time is 258 milliseconds. Since we must both swap out and swap in, the total swap time is about 516 milliseconds.

For efficient CPU utilization, we want the 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 0.516 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 512 MB of main memory and a resident operating system taking 25 MB, the maximum size of the user process is 487 MB. However, many user processes may be much smaller than this—say, 10 MB. A 10-MB process could be swapped out in 258 milliseconds, compared with the 6.4 seconds required for swapping 256 MB. Clearly, 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 method 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.

Swapping is constrained by other factors as well. If we want to swap a process, we must be sure that it is completely idle. Of particular concern is any pending I/O. A process may be waiting for an I/O operation when we want to swap that process to free up 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 is queued because the device is busy. 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. There are two main solutions to this problem: Never swap a process with pending I/O, or execute I/O operations only into operating-system buffers.

Transfers between operating-system buffers and process memory then occur only when the process is swapped in.

The assumption, mentioned earlier, 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 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 is normally disabled but will start if many processes are running and are using a threshold amount of memory. Swapping is again halted when the load on the system is reduced. Memory management in UNIX is described fully in Sections 21.7 and A.6.

Early PCs—which lacked the sophistication to implement more advanced memory-management methods—ran multiple large processes by using a modified version of swapping. A prime example is the Microsoft Windows 3.1 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 preempt one process for another. Any swapped-out process remains swapped out (and not executing) until the user selects that process to run. Subsequent versions of Microsoft operating systems take advantage of the advanced MMU features now found in PCs. We explore such features in Section 8.4 and in Chapter 9, where we cover virtual memory.

8.3 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate the parts of the main memory in the most efficient way possible. This section explains one common method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can 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, programmers usually place the operating system in low memory as well. Thus, in this text, we discuss only the situation where the operating system resides in low memory. The development of the other situation is similar.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In this **contiguous memory allocation**, each process is contained in a single contiguous section of memory.

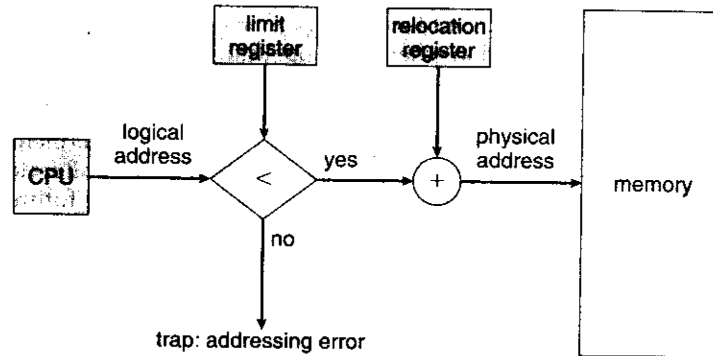


Figure 8.6 Hardware support for relocation and limit registers.

8.3.1 Memory Mapping and Protection

Before discussing memory allocation further, we must discuss the issue of memory mapping and protection. We can provide these features by using a relocation register, as discussed in Section 8.1.3, with a limit register, as discussed in Section 8.1.1. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). 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 against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

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 conta. .s code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, we do not want 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.

8.3.2 Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this **multiple-partition method**, 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 method was originally

used by the IBM OS/360 operating system (called MFT); it is no longer in use. The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in batch environments. Many of the ideas presented here are also applicable to a time-sharing environment in which pure segmentation is used for memory management (Section 8.6).

In the fixed-partition scheme, 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 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.

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 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—that is, no available block of memory (or 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, at any given time we have a *set* of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. 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, these adjacent holes are merged to form one larger hole. At this point, the system 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 concerns how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly 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 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 time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

8.3.3 Fragmentation

Both the first-fit and best-fit strategies for memory allocation 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 there is enough total memory space to satisfy a request, but the available spaces are not contiguous; storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, whereas 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 algorithm is used, external fragmentation will be a problem.

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

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a 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 to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process 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 so as to place all free memory together in one large block. Compaction is not always possible, however. 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 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 expensive.

Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. Two complementary techniques achieve this solution: paging (Section 8.4) and segmentation (Section 8.6). These techniques can also be combined (Section 8.7).

8.4 Paging

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory-management schemes used before the introduction of paging suffered from this problem. The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store also has the fragmentation problems discussed in connection with main memory, except that access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is commonly used in most operating systems.

Traditionally, support for paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and operating system, especially on 64-bit microprocessors.

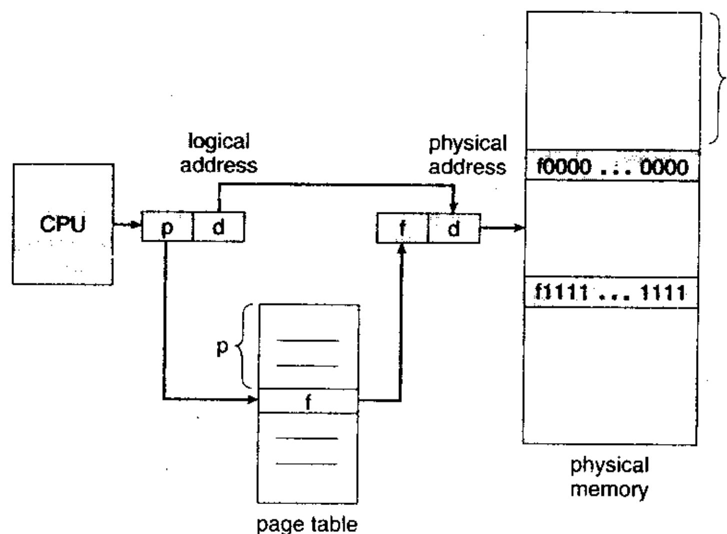


Figure 8.7 Paging hardware.

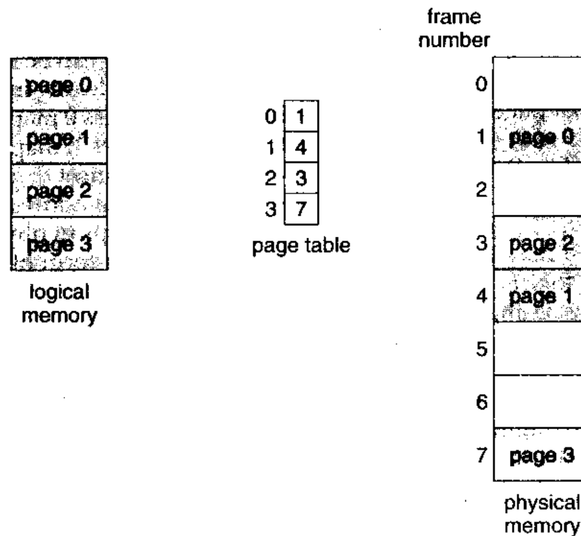


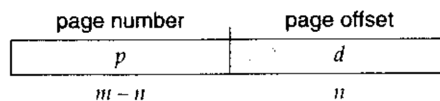
Figure 8.8 Paging model of logical and physical memory.

8.4.1 Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory 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 8.7. 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 memory address that is sent to the memory unit. The paging model of memory is shown in Figure 8.8.

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 16 MB 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:



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

As a concrete (although minuscule) example, consider the memory in Figure 8.9. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of 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.

You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

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 coincide with page boundaries, the *last* frame allocated may not be completely full. For example, if page size

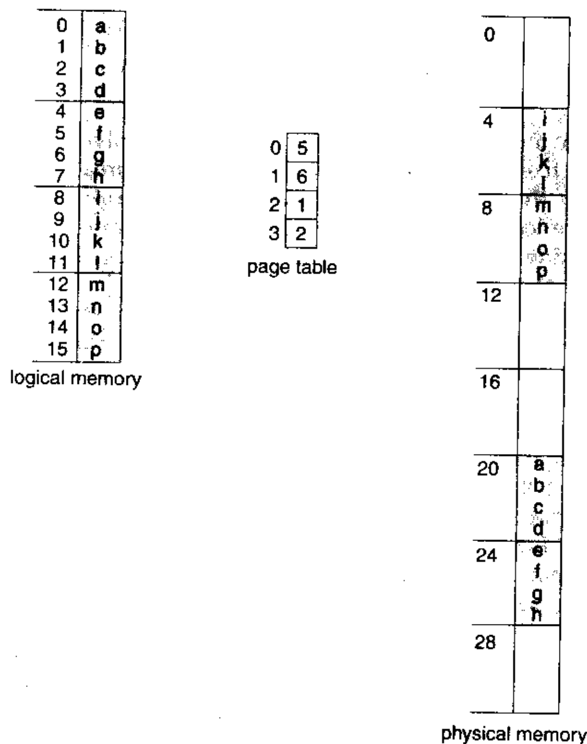


Figure 8.9 Paging example for a 32-byte memory with 4-byte pages.

is 2,048 bytes, a process of 72,766 bytes would need 35 pages plus 1,086 bytes. It would be allocated 36 frames, resulting in an internal fragmentation of 2,048 − 1,086 = 962 bytes. In the worst case, a process would need n pages plus 1 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, overhead is 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 between 4 KB and 8 KB in size, and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages. Researchers are now developing variable on-the-fly page-size support.

Usually, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 2^{32} physical page frames. If frame size is 4 KB, then a system with 4-byte entries can address 2^{44} bytes (or 16 TB) of physical memory.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving 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 8.10).

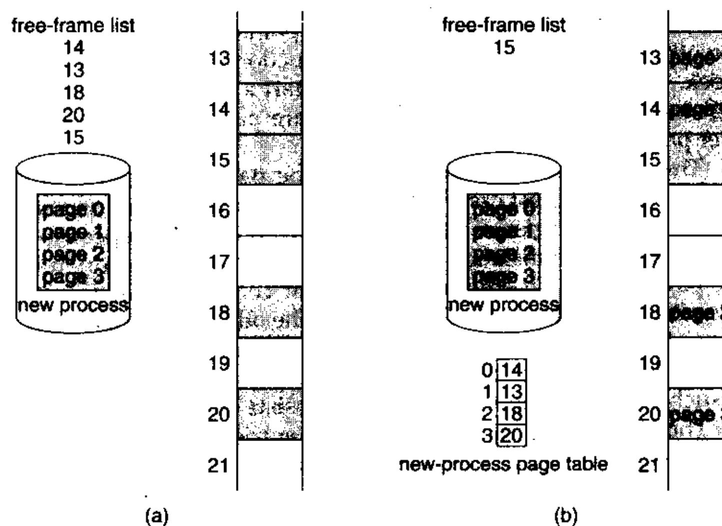


Figure 8.10 Free frames (a) before allocation and (b) after allocation.

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 memory as one single 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.

Since 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.

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.

8.4.2 Hardware Support

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.

The hardware implementation of the page table can be done in several 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 8 KB. 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 *ch8/8*. 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, called a **translation look-aside buffer (TLB)**. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware, however, is expensive. Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, 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 if an unmapped memory reference were used.

If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory (Figure 8.11). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random. Furthermore, some TLBs allow entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down.

Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

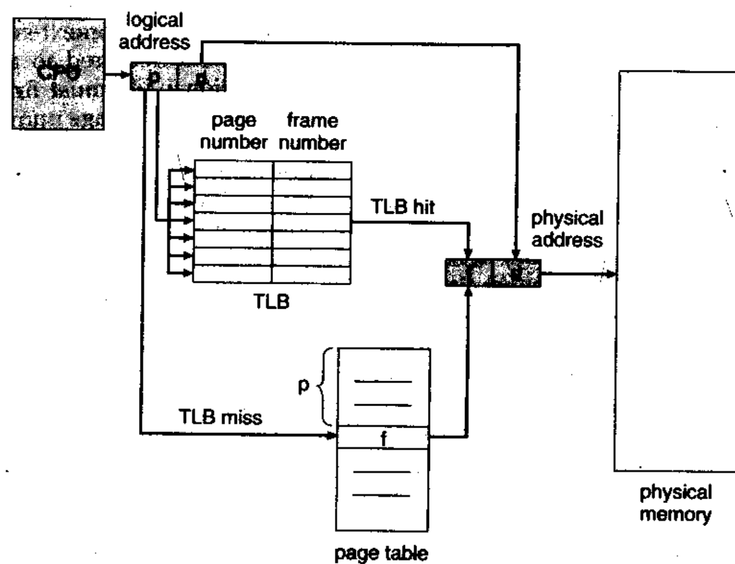


Figure 8.11 Paging hardware with TLB.

The percentage of times that a particular page number is found in the TLB is called the **hit ratio**. An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time. If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB (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 weight 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 access time. We will further explore the impact of the hit ratio on the TLB in Chapter 9.

8.4.3 Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.

One bit can define a page to be read–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 (or memory-protection violation).

We can easily expand this approach 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, we can allow any combination of these accesses. Illegal attempts will be trapped to the operating system.

One additional bit is generally attached to each entry in the page table: a **valid–invalid** bit. When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to "invalid," the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we get the situation shown in Figure 8.12. 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, will find that the valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

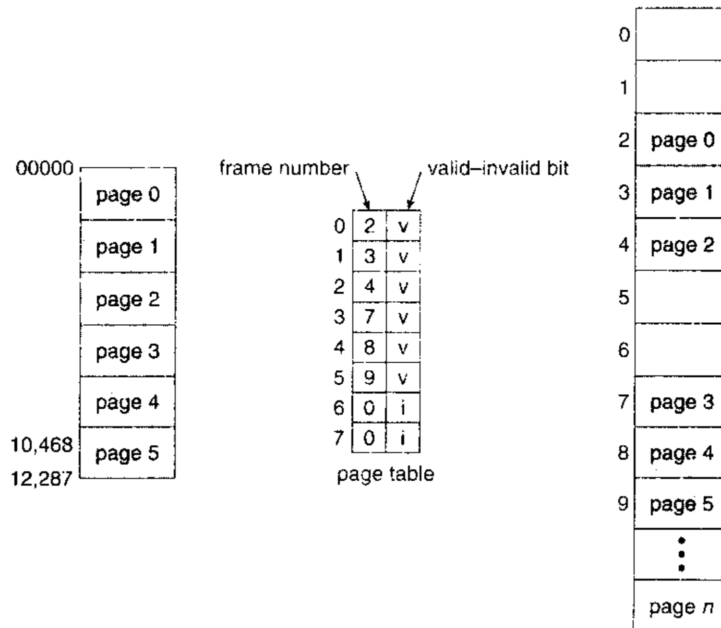


Figure 8.12 Valid (v) or invalid (i) bit in a page table.

Notice that this scheme has created a problem. Because the program extends to only address 10468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.

Rarely does a process use all 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.

8.4.4 Shared Pages

An 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 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is **reentrant code** (or **pure code**), however, it can be shared, as shown in Figure 8.13. Here we see a three-page editor—each

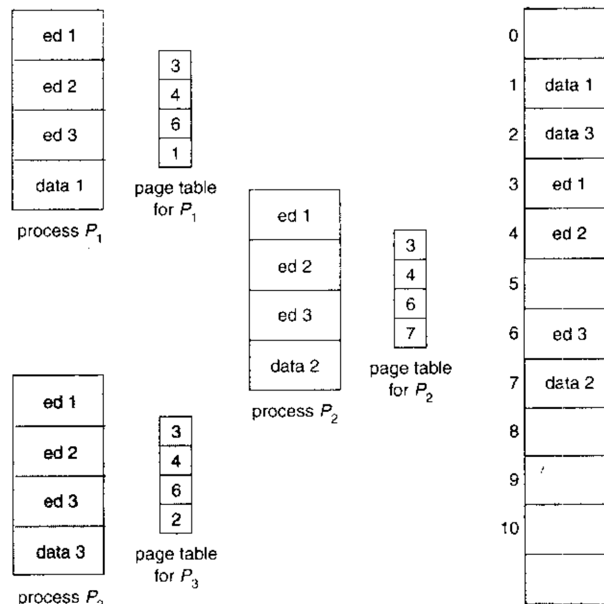


Figure 8.13 Sharing of code in a paging environment.

page 50 KB in size (the large page size is used to simplify the figure)—being shared among three processes. Each process has its own data page.

Reentrant code is non-self-modifying code; 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's execution. The data for two different processes will, of course, be different.

Only one copy of the editor need 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 (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings.

Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be 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.

The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads, described in Chapter 4. Furthermore, recall that in Chapter 3 we described shared memory as a method of interprocess communication. Some operating systems implement shared memory using shared pages.

Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages. We will cover several other benefits in Chapter 9.

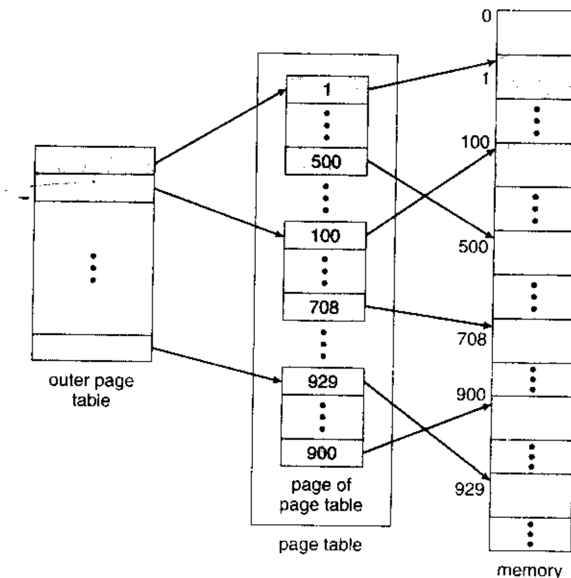


Figure 8.14 A two-level page-table scheme.

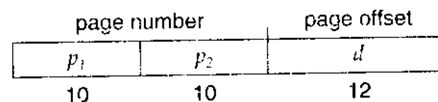
8.5

In this section, we explore some of the most common techniques for structuring the page table.

8.5.1 Hierarchical Paging

Most modern computer systems support a 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 4 KB (2^{12}), then a page table may consist of up to 1 million entries ($2^{32}/2^{12}$). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB 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 problem is to divide the page table into smaller pieces. We can accomplish this division in several ways.

One way is to use a two-level paging algorithm, in which the page table itself is also paged (Figure 8.14). Remember our example of a 32-bit machine with a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because 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:



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 method for this architecture is shown in Figure 8.15. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**.

The VAX architecture also supports a variation of two-level paging. The VAX is a 32-bit machine with a 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

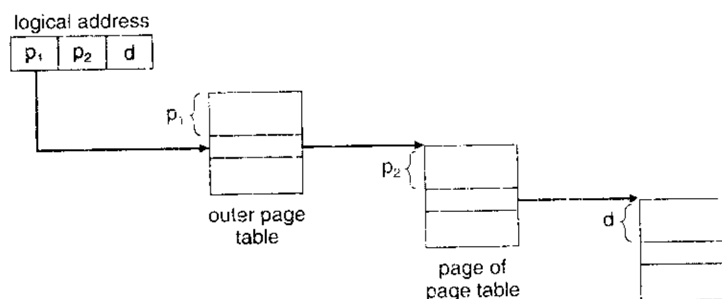


Figure 8.15 Address translation for a two-level 32-bit paging architecture.

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 final 9 bits represent an offset in the desired page. By partitioning the page table in this manner, the operating system can 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. Even when this scheme is used, the size of a one-level page table for a VAX process using one section is 2^{21} bits $*$ 4 bytes per entry = 8 MB. So that main-memory use is reduced further, 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, let us suppose that the page size in such a system is 4 KB (2^{12}). In this case, the page table consists of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses look like this:

outer page	inner page	offset
p_1	p_2	d
42	10	12

The outer page table consists of 2^{42} entries, or 2^{44} bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. This approach is also used on some 32-bit processors for added flexibility and efficiency.

We can divide the outer page table in various ways. We can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes); a 64-bit address space is still daunting:

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

The outer page table is still 2^{34} bytes in size.

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.

For 64-bit architectures, hierarchical page tables are generally considered inappropriate. For example, the 64-bit UltraSPARC would require seven levels of

paging—a prohibitive number of memory accesses—to translate each logical address.

8.5.2 Hashed Page Tables

A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 8.16.

A variation of this scheme that is favorable for 64-bit address spaces has been proposed. This variation uses **clustered page tables**, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.

8.5.3 Inverted Page Tables

Usually, each process has an associated page table. 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

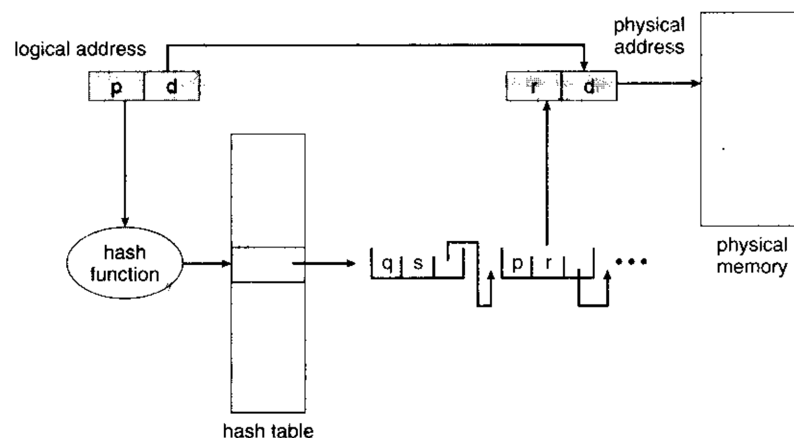


Figure 8.16 Hashed page table.

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 method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how 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 (or 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, only one page table is in the system, and it has only one entry for each page of physical memory. Figure 8.17 shows the operation of an inverted page table. Compare it with Figure 8.7, which depicts a standard page table in operation. Inverted page tables often require that an address-space identifier (Section 8.4.2) be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory. Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame. Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.

To illustrate this method, we describe a simplified version 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> where the process-id assumes the role of the address-space identifier. 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

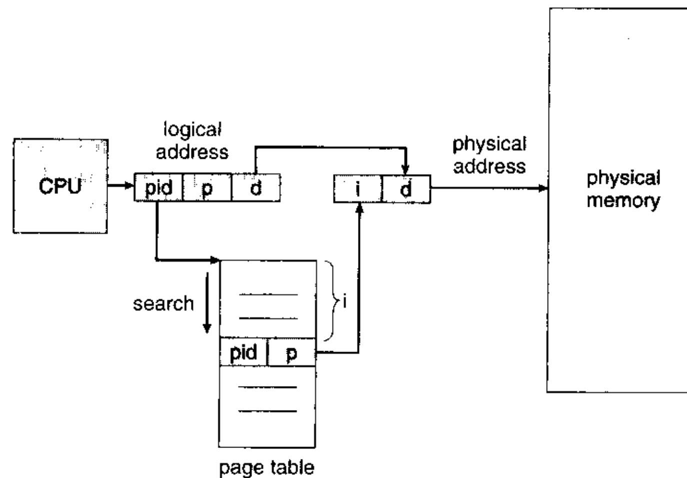


Figure 8.17 Inverted page table.

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.

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 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, as described in Section 8.5.2, 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, recall that the TLB is searched first, before the hash table is consulted.

Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is usually implemented as multiple virtual addresses (one for each process sharing the memory) that are mapped to one physical address. This standard method cannot be used with inverted page tables; because there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses. A simple technique for addressing this issue is to allow the page table to contain only one mapping of a virtual address to the shared physical address. This means that references to virtual addresses that are not mapped result in page faults.

8.6

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. As we have already seen, the user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. This mapping allows differentiation between logical memory and physical memory.

8.6.1 Basic Method

Do users think of memory as a linear array of bytes, some containing instructions and others containing data? Most people would say no. Rather, users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments (Figure 8.18).

Consider how you think of a program when you are writing it. You think of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. You talk about "the stack," "the math library," "the main program," without caring what addresses in memory these elements occupy. You are not concerned with whether the stack 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

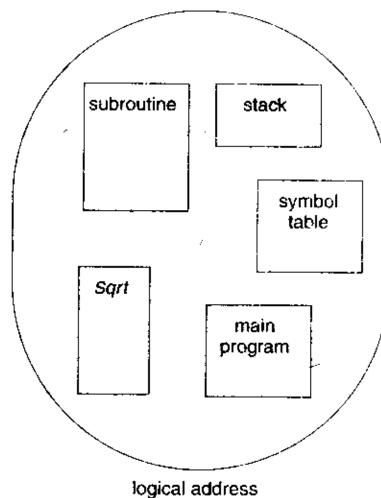


Figure 8.18 User's view of a program.

of the program, the seventh stack frame entry in the stack, the fifth instruction of the Sqrt(), 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 segment has a name and a length. The 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 the paging scheme, in which the user specifies only a single address, which is 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 C compiler might create separate segments for the following:

The code

Global variables

The heap, from which memory is allocated

The stacks used by each thread

The standard C library

Libraries that are linked in during compile time 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 in 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 8.19. 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 to 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). When an 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 8.20. 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 memory (or base) and the length of that segment (or 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 1,000 bytes long.

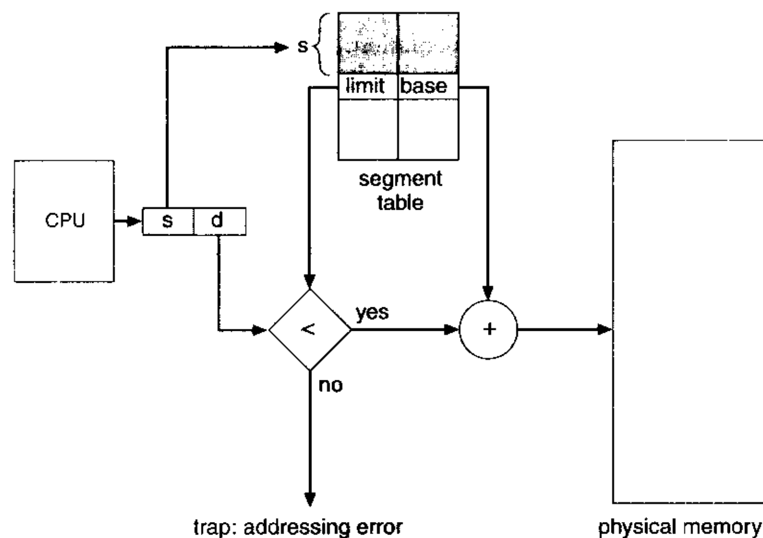


Figure 8.19 Segmentation hardware.

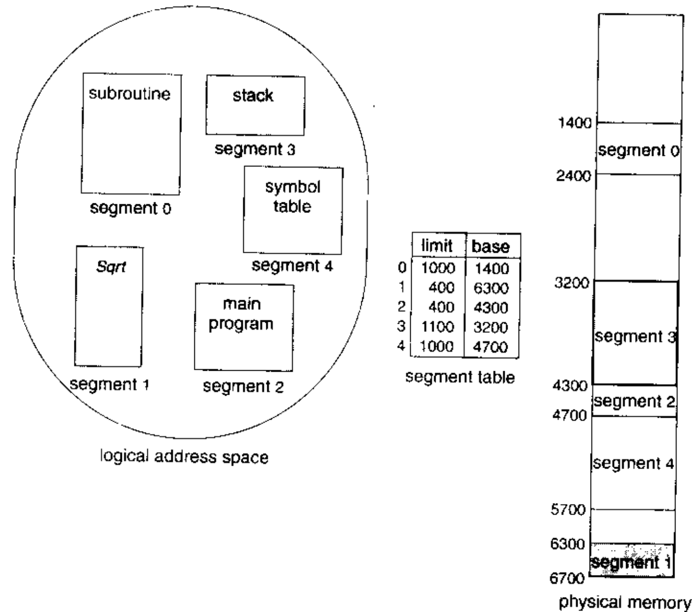


Figure 8.20 Example of segmentation.

8.7

Both paging and segmentation have advantages and disadvantages. In fact, some architectures provide both. In this section, we discuss the Intel Pentium architecture, which supports both pure segmentation and segmentation with paging. We do not give a complete description of the memory-management structure of the Pentium in this text. Rather, we present the major ideas on which it is based. We conclude our discussion with an overview of Linux address translation on Pentium systems.

In Pentium systems, the CPU generates logical addresses, which are given to the segmentation unit. The segmentation unit produces a linear address for each logical address. The linear address is then given to the paging unit, which in turn generates the physical address in main memory. Thus, the segmentation and paging units form the equivalent of the memory-management unit (MMU). This scheme is shown in Figure 8.21.

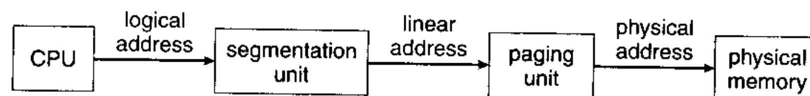


Figure 8.21 Logical to physical address translation in the Pentium.

8.7.1 Pentium Segmentation

The Pentium architecture allows a segment to be as large as 4 GB, and the maximum number of segments per process is 16 KB. The logical-address space of a process is divided into two partitions. The first partition consists of up to 8 KB segments that are private to that process. The second partition consists of up to 8 KB 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 consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit 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 (or 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 also has six 8-byte microprogram registers to hold the corresponding descriptors from either the LDT or GDT. This cache lets the Pentium avoid having to read the descriptor from memory for every memory reference.

The linear address on the Pentium is 32 bits long and is formed as follows. The segment register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment in question is 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 is shown in Figure 8.22. In

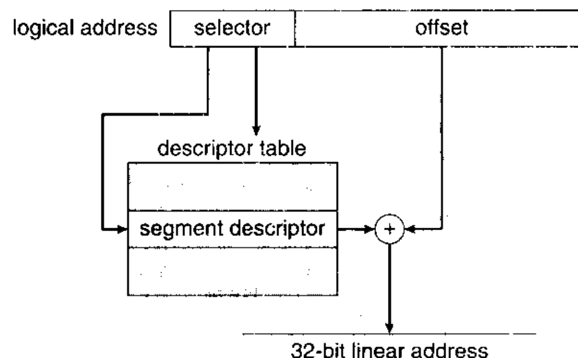
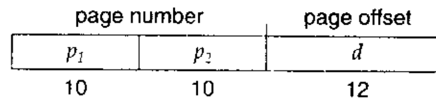


Figure 8.22 Intel Pentium segmentation.

the following section, we discuss how the paging unit turns this linear address into a physical address.

8.7.2 Pentium Paging

The Pentium architecture allows a page size of either 4 KB or 4 MB. For 4-KB pages, the Pentium uses a two-level paging scheme in which the division of the 32-bit linear address is as follows:



The address-translation scheme for this architecture is similar to the scheme shown in Figure 8.15. The Intel Pentium address translation is shown in more detail in Figure 8.23. The ten high-order bits reference an entry in the outermost page table, which the Pentium terms the **page directory**. (The CR3 register points to the page directory for the current process.) The page directory entry points to an inner page table that is indexed by the contents of the innermost ten bits in the linear address. Finally, the low-order bits 0–11 refer to the offset in the 4-KB page pointed to in the page table.

One entry in the page directory is the Page Size flag, which—if set—indicates that the size of the page frame is 4 MB and not the standard 4 KB. If this flag is set, the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.

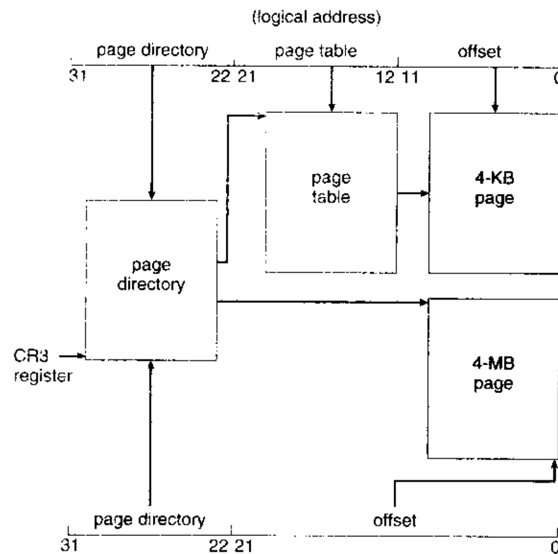


Figure 8.23 Paging in the Pentium architecture.

To improve the efficiency of physical memory use, Intel Pentium 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.

8.7.3 Linux on Pentium Systems

As an illustration, consider the Linux operating system running on the Intel Pentium architecture. Because Linux is designed to run on a variety of processors—many of which may provide only limited support for segmentation—Linux does not rely on segmentation and uses it minimally. On the Pentium, Linux uses only six segments:

- A segment for kernel code
- A segment for kernel data
- A segment for user code
- A segment for user data
- A task-state segment (TSS)
- A default LDT segment

The segments for user code and user data are shared by all processes running in user mode. This is possible because all processes use the same logical address space and all segment descriptors are stored in the global descriptor table (GDT). Furthermore, each process has its own task-state segment (TSS), and the descriptor for this segment is stored in the GDT. The TSS is used to store the hardware context of each process during context switches. The default LDT segment is normally shared by all processes and is usually not used. However, if a process requires its own LDT, it can create one and use that instead of the default LDT.

As noted, each segment selector includes a 2-bit field for protection. Thus, the Pentium allows four levels of protection. Of these four levels, Linux only recognizes two: user mode and kernel mode.

Although the Pentium uses a two-level paging model, Linux is designed to run on a variety of hardware platforms, many of which are 64-bit platforms where two-level paging is not plausible. Therefore, Linux has adopted a three-level paging strategy that works well for both 32-bit and 64-bit architectures.

The linear address in Linux is broken into the following four parts:

global directory	middle directory	page table	offset

Figure 8.24 highlights the three-level paging model in Linux.

The number of bits in each part of the linear address varies according to architecture. However, as described earlier in this section, the Pentium architecture only uses a two-level paging model. How, then, does Linux apply

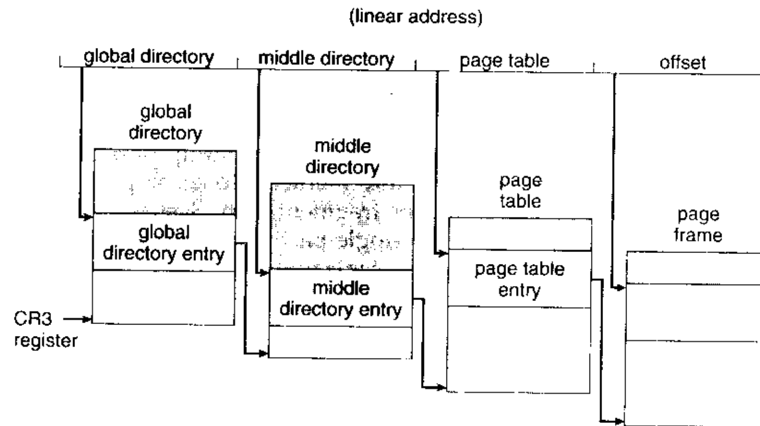


Figure 8.24 Three-level paging in Linux.

its three-level model on the Pentium? In this situation, the size of the middle directory is zero bits, effectively bypassing the middle directory.

Each task in Linux has its own set of page tables and—just as in Figure 8.23—the CR3 register points to the global directory for the task currently executing. During a context switch, the value of the CR3 register is saved and restored in the TSS segments of the tasks involved in the context switch.

8.8

Memory-management algorithms for multiprogrammed operating systems range from the simple single-user system approach to paged segmentation. The most important 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 various memory-management algorithms (contiguous allocation, paging, segmentation, and combinations of paging and segmentation) differ in many aspects. In comparing different memory-management strategies, we use the following considerations:

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

Performance. As the memory-management algorithm becomes more complex, the time required to map a logical address to a physical address increases. For the simple systems, we need only compare or add to the logical address—operations that are fast. Paging and segmentation can be as fast if the mapping table is implemented in fast registers. If the table is

in memory, however, user memory accesses can be degraded substantially. A TLB can reduce the performance degradation to an acceptable level.

Fragmentation. A multiprogrammed system will generally perform more efficiently if it has 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 in such a way that the program does not notice 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. Swapping can be added to any algorithm. 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.

8.1 Explain the difference between internal and external fragmentation.

8.2 Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linkage editor is used to combine multiple object modules into a single program binary. How does the linkage editor change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linkage editor to facilitate the memory binding tasks of the linkage editor?

8.3 Most systems allow programs to allocate more memory to its address space during execution. Data allocated in the heap segments of programs

is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?

a. contiguous-memory allocation

b. pure segmentation

c. pure paging

8.4 Compare the main memory organization schemes of contiguous-memory allocation, pure segmentation, and pure paging with respect to the following issues:

a. external fragmentation

b. internal fragmentation

c. ability to share code across processes

8.5 Compare paging with segmentation with respect to the amount of memory required by the address translation structures in order to convert virtual addresses to physical addresses.

8.6 Program binaries in many systems are typically structured as follows. Code is stored starting with a small fixed virtual address such as 0. The code segment is followed by the data segment that is used for storing the program variables. When the program starts executing, the stack is allocated at the other end of the virtual address space and is allowed to grow towards lower virtual addresses. What is the significance of the above structure on the following schemes?

a. contiguous-memory allocation

b. pure segmentation

c. pure paging

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

8.8 Explain why sharing a reentrant module is easier when segmentation is used than when pure paging is used.

8.9 What is the purpose of paging the page tables?

8.10 Consider the hierarchical paging scheme used by the VAX architecture. How many memory operations are performed when an user program executes a memory load operation?

8.11 Consider the Intel address-translation scheme shown in Figure 8.22.

a. Describe all the steps taken by the Intel Pentium 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?

c. Are there any disadvantages to this address-translation system? If so, what are they? If not, why is it not used by every manufacturer?

Bibliographical 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. Knuth [1973] discussed the 50-percent rule.

The concept of paging can be credited to the designers of the Atlas system, which has been described by Kilburn et al. [1961] and by 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] and Daley and Dennis [1967]).

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

Address translation in software is covered in Jacob and Mudge [1997].

Hennessy and Patterson [2002] discussed the hardware aspects of TLBs, caches, and MMUs. Talluri et al. [1995] discusses page tables for 64-bit address spaces. Alternative approaches to enforcing memory protection are proposed and studied in Wahbe et al. [1993a], Chase et al. [1994], Bershad et al. [1995], and Thorn [1997]. Dougan et al. [1999] and Jacob and Mudge [2001] discuss techniques for managing the TLB. Fang et al. [2001] evaluate support for large pages.

Tanenbaum [2001] discusses Intel 80386 paging. Memory management for several architectures—such as the Pentium II, PowerPC, and UltraSPARC—was described by Jacob and Mudge [1998a]. Segmentation on Linux systems is presented in Bovet and Cesati [2002]