

# Deadlocks



In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a 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. Although some applications can identify programs that may deadlock, operating systems typically do not provide deadlock-prevention facilities, and it remains the responsibility of programmers to ensure that they design deadlock-free programs. 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.

## CHAPTER OBJECTIVES

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks.
- To present a number of different methods for preventing or avoiding deadlocks in a computer system.

### 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 a separate resource class 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.** The process requests the resource. 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 `acquire()` and `release()` operations on semaphores or through acquisition and release of an object's lock via Java's `synchronized` keyword. 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 **deadlocked 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 deadlocked state, consider a system with three CD read-write (RW) drives. Suppose each of three processes holds one of these drives. If each process now requests another drive, the three processes will be in a deadlocked 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.

## 7.2 Deadlock Characterization

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  $\{P_0, P_1, \dots, P_n\}$  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 of nodes:  $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the active processes in the system, and  $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system.

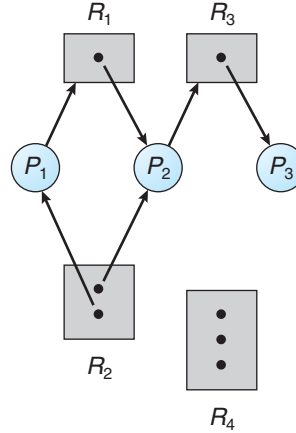
A directed edge from process  $P_i$  to resource type  $R_j$  is denoted by  $P_i \rightarrow R_j$ ; it signifies that process  $P_i$  has requested an instance of resource type  $R_j$  and is currently waiting for that resource. A directed edge from resource type  $R_j$  to process  $P_i$  is denoted 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 7.1 depicts the following situation.

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



**Figure 7.1** Resource-allocation graph.

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

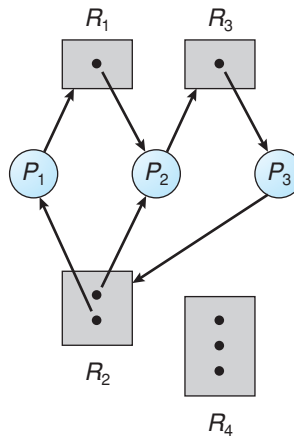
$$\begin{aligned} P_1 &\rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \\ P_2 &\rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2 \end{aligned}$$

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 7.3. In this example, we also have a cycle:

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

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



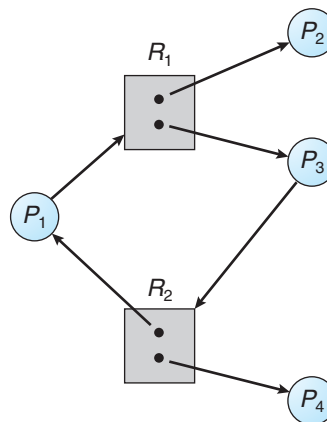
**Figure 7.2** Resource-allocation graph with a deadlock.

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.

#### 7.2.2.1 Deadlock in a Multithreaded Java Program

Before we proceed to a discussion of handling deadlocks, let's see how deadlock can occur in a multithreaded Java program, as shown in Figure 7.4. In this example, we have two threads—threadA and threadB—as well as two reentrant locks—first and second. (Recall from Chapter 6 that a reentrant lock acts as a simple mutual exclusion lock.) In this example, threadA attempts to acquire the locks in the order (1) first, (2) second; threadB attempts using the order (1) second, (2) first. Deadlock is possible in the following scenario:

threadA  $\rightarrow$  second  $\rightarrow$  threadB  $\rightarrow$  first  $\rightarrow$  threadA



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

```

class A implements Runnable
{
    private Lock first, second;

    public A(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }

    public void run() {
        try {
            first.lock();
            // do something
            second.lock();
            // do something else
        }
        finally {
            first.unlock();
            second.unlock();
        }
    }
}

class B implements Runnable
{
    private Lock first, second;

    public A(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }

    public void run() {
        try {
            second.lock();
            // do something
            first.lock();
            // do something else
        }
        finally {
            second.unlock();
            first.unlock();
        }
    }
}

public class DeadlockExample {
    // Figure 7.5
}

```

**Figure 7.4** Deadlock example.

```

public static void main(String arg[]) {
    Lock lockX = new ReentrantLock();
    Lock lockY = new ReentrantLock();

    Thread threadA = new Thread(new A(lockX,lockY));
    Thread threadB = new Thread(new B(lockX,lockY));

    threadA.start();
    threadB.start();
}

```

**Figure 7.5** Creating the threads (continuation of Figure 7.4).

Note that, even though deadlock is possible, it will not occur if `threadA` is able to acquire and release the locks for `first` and `second` before `threadB` 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.

## 7.3 Methods for Handling Deadlocks

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 deadlocked state.
- We can allow the system to enter a deadlocked 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. The JVM also does nothing to manage deadlocks. It is up to the application developer to write programs that handle deadlocks on these systems.

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. 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.

### 7.3.1 Three Major Methods

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.

In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which 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.

### 7.3.2 Handling Deadlocks in Java

As noted earlier, the JVM does nothing to manage deadlocks; it is up to the application developer to write programs that are deadlock-free. In the remainder of this section, we illustrate how deadlock is possible when selected methods of the core Java API are used and how the programmer can develop programs that appropriately handle deadlock.

In Chapter 4, we introduced Java threads and some of the API that allows users to create and manipulate threads. Two additional methods of the `Thread` class are the `suspend()` and `resume()` methods, which were deprecated in later versions of the Java API because they could lead to deadlock. (A deprecated method indicates that the method is still part of the Java API, but its use is discouraged.)

The `suspend()` method suspends execution of the thread currently running. The `resume()` method resumes execution of a suspended thread. Once

a thread has been suspended, it can continue only if another thread resumes it. Furthermore, a suspended thread continues to hold all locks while it is blocked. Deadlock is possible if a suspended thread holds a lock on an object and the thread that can resume it must own this lock before it can resume the suspended thread.

```
import java.applet.*;
import java.awt.*;

public class ClockApplet extends Applet implements Runnable
{
    private Thread clockThread;
    private boolean ok = false;
    private Object mutex = new Object();

    public void run() {
        while (true) {
            try {
                // sleep for 1 second
                Thread.sleep(1000);

                // repaint the date and time
                repaint();

                // see if we need to suspend ourself
                synchronized (mutex) {
                    while (ok == false)
                        mutex.wait();
                }
            }
            catch (InterruptedException e) { }
        }
    }

    public void start() {
        // Figure 7.7
    }

    public void stop() {
        // Figure 7.7
    }

    public void paint(Graphics g) {
        g.drawString(new java.util.Date().toString(),10,30);
    }
}
```

**Figure 7.6** Applet that displays the date and time of day.

Another method, `stop()`, has been deprecated as well, but not because it can lead to deadlock. Unlike the situation in which a thread has been suspended, when a thread has been stopped, it releases all the locks that it owns. However, locks are generally used in the following progression: (1) acquire the lock, (2) access a shared data structure, and (3) release the lock. If a thread is in the middle of step 2 when it is stopped, it will release the lock; but it may leave the shared data structure in an inconsistent state. In Section 4.5.2, we discussed how to terminate a thread using deferred cancellation rather than asynchronously canceling a thread using the `stop()` method. Here, we present a strategy for suspending and resuming a thread without using the deprecated `suspend()` and `resume()` methods.

The program shown in Figure 7.6 is a multithreaded applet that displays the time of day. When this applet starts, it creates a second thread (which we will call the *clock thread*) that outputs the time of day. The `run()` method of the clock thread alternates between sleeping for one second and then calling the `repaint()` method. The `repaint()` method ultimately calls the `paint()` method, which draws the current date and time in the browser's window.

This applet is designed so that the clock thread is running while the applet is visible; if the applet is not being displayed (as when the browser window

```

/**
 * This method is called when the applet is
 * started or we return to the applet.
 */
public void start() {
    ok = true;

    if (clockThread == null) {
        clockThread = new Thread(this);
        clockThread.start();
    }
    else {
        synchronized(mutex) {
            mutex.notify();
        }
    }
}

/**
 * This method is called when we
 * leave the page the applet is on.
 */
public void stop() {
    synchronized(mutex) {
        ok = false;
    }
}

```

**Figure 7.7** `start()` and `stop()` methods for the applet (continuation of Figure 7.6).

has been minimized), the clock thread is suspended from execution. This is accomplished by overriding the `start()` and `stop()` methods of the `Applet` class. (Be careful not to confuse these with the `start()` and `stop()` methods of the `Thread` class.) The `start()` method of an applet is called when an applet is first created. If the user leaves the web page, if the applet scrolls off the screen, or if the browser window is minimized, the applet's `stop()` method is called. If the user returns to the applet's web page, the applet's `start()` method is called again.

The applet uses the Boolean variable `ok` to indicate whether the clock thread can run. This variable will be set to `true` in the `start()` method of the applet, indicating that the clock thread can run. The `stop()` method of the applet will set it to `false`. The clock thread will check the value of this Boolean variable in its `run()` method and will only proceed if it is `true`. Because the thread for the applet and the clock thread will be sharing this variable, access to it will be controlled through a synchronized block. This program is shown in Figure 7.7.

If the clock thread sees that the Boolean value is `false`, it suspends itself by calling the `wait()` method for the object `mutex`. When the applet wishes to resume the clock thread, it sets the Boolean variable to `true` and calls `notify()` for the `mutex` object. This call to `notify()` awakens the clock thread. It checks the value of the Boolean variable and, seeing that it is now `true`, proceeds in its `run()` method, displaying the date and time.

## 7.4 Deadlock Prevention

As we noted in Section 7.2.1, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock. 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 our example, 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. Otherwise, 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 the process is currently holding 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 receiving 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  $R = \{R_1, R_2, \dots, R_m\}$  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:

$$\begin{aligned} F(\text{tape drive}) &= 1 \\ F(\text{disk drive}) &= 5 \\ F(\text{printer}) &= 12 \end{aligned}$$

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  $F(R_j) > F(R_i)$ . 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 a process requesting an instance of resource type  $R_j$  must have released any resources  $R_i$  such that  $F(R_i) \geq F(R_j)$ . Note also that if several instances of the same resource type are needed, a *single* request for all of them must be issued.

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  $\{P_0, P_1, \dots, P_n\}$ , 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  $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$ . 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 Java program shown in Figure 7.4 was

$$\begin{aligned} F(\text{first}) &= 1 \\ F(\text{second}) &= 5 \end{aligned}$$

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

Keep in mind that developing an ordering, or hierarchy, does not in itself 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 specify that  $F(\text{tape drive}) < F(\text{printer})$ .

To develop a lock ordering, Java programmers are encouraged to use the method `System.identityHashCode()`, which returns the value that would be returned by the default value of the object's `hashCode()` method. For example, to obtain the value of `identityHashCode()` for the first and second locks in the Java program shown in Figure 7.4, you would use the following statements:

```
int firstOrderingValue = System.identityHashCode(first);
int secondOrderingValue = System.identityHashCode(second);
```

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 7.4 as an example. Assume that `threadA` is the first to acquire the locks and does so in the order (1) first, (2) second. Witness records the relationship that first must be acquired before second. If `threadB` later attempts to acquire the locks out of order, witness generates a warning message on the system console.

Finally, it is important to note that imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically. For example, assume we have a method that transfers funds between two accounts. To prevent a race condition, we use the object lock associated with each `Account` object in a synchronized block. The code appears as follows:

```
void transaction(Account from, Account to, double amount) {
    synchronized(from) {
        synchronized(to) {
            from.withdraw(amount);
            to.deposit(amount);
        }
    }
}
```

Deadlock is possible if two threads simultaneously invoke the `transaction()` method, transposing different accounts. That is, one thread might invoke

```
transaction(checkingAccount, savingsAccount, 25);
```

and another might invoke

```
transaction(savingsAccount, checkingAccount, 50);
```

We leave it as an exercise for the reader to figure out a solution to fix this situation.



## 7.5 Deadlock Avoidance

Deadlock-prevention algorithms, as discussed in Section 7.4, prevent deadlocks by limiting how requests can be made. The limits 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.

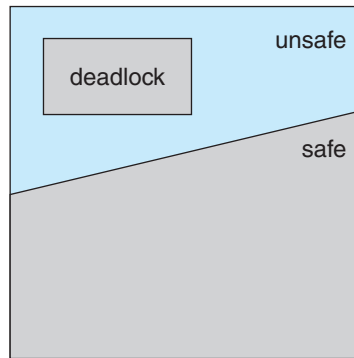
The various algorithms that use this approach differ in the amount and type of information required. The simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm 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  $\langle P_1, P_2, \dots, P_n \rangle$  is a safe sequence for the current allocation state if, for each  $P_i$ , the resource requests that  $P_i$  can 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 7.8). 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 in such a way that a deadlock occurs. The behavior of the processes controls unsafe states.





**Figure 7.8** Safe, unsafe, and deadlocked state spaces.

To illustrate, we consider a system with twelve magnetic tape drives and three processes:  $P_0$ ,  $P_1$ , and  $P_2$ . Process  $P_0$  requires ten tape drives, process  $P_1$  may need as many as four tape drives, and process  $P_2$  may need up to nine tape drives. Suppose that, at time  $t_0$ , process  $P_0$  is holding five tape drives, process  $P_1$  is holding two tape drives, and process  $P_2$  is holding two tape drives. (Thus, there are three 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  $\langle P_1, P_0, P_2 \rangle$  satisfies the safety condition. Process  $P_1$  can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process  $P_0$  can get all its tape drives and return them (the system will then have ten available tape drives); and finally process  $P_2$  can get all its tape drives and return them (the system will then have all twelve 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 four available tape drives. Since process  $P_0$  is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process  $P_2$  may request six additional tape drives and, if it does, will 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

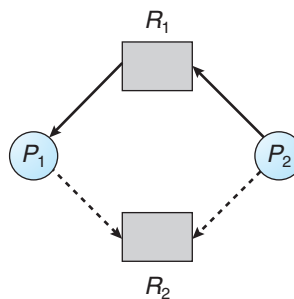
If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph defined in Section 7.2.2 for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called a **claim edge**. A claim edge  $P_i \rightarrow R_j$  indicates that process  $P_i$  may request resource  $R_j$  at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process  $P_i$  requests resource  $R_j$ , the claim edge  $P_i \rightarrow R_j$  is converted to a request edge. Similarly, when a resource  $R_j$  is released by  $P_i$ , the assignment edge  $R_j \rightarrow P_i$  is reconverted to a claim edge  $P_i \rightarrow R_j$ .

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

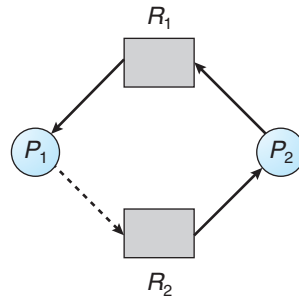
Now suppose that process  $P_i$  requests resource  $R_j$ . The request can be granted only if converting the request edge  $P_i \rightarrow R_j$  to an assignment edge  $R_j \rightarrow P_i$  does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a 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. In that case, 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 7.9. 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



**Figure 7.9** Resource-allocation graph for deadlock avoidance.



**Figure 7.10** An unsafe state in a resource-allocation graph.

7.10). A cycle, as mentioned, indicates that the system is in an unsafe state. If  $P_1$  requests  $R_2$ , and  $P_2$  requests  $R_1$ , then a deadlock will occur.

### 7.5.3 Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the *banker's algorithm*. The name was 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. We need the following data structures, where  $n$  is the number of processes in the system and  $m$  is the number of resource types:

- **Available.** A vector of length  $m$  indicates the number of available resources of each type. If  $Available[j]$  equals  $k$ , then  $k$  instances of resource type  $R_j$  are 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  $i = 1, 2, \dots, n$ . For example, if  $X = (1, 7, 3, 2)$  and  $Y = (0, 3, 2, 1)$ , then  $Y \leq X$ . In addition,  $Y < X$  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:

1. Let *Work* and *Finish* be vectors of length  $m$  and  $n$ , respectively. Initialize  $Work = Available$  and  $Finish[i] = false$  for  $i = 0, 1, \dots, n - 1$ .
2. Find an index  $i$  such that both
  - a.  $Finish[i] == false$
  - b.  $Need_i \leq Work$
 If no such  $i$  exists, go to step 4.
3.  $Work = Work + Allocation_i$   
 $Finish[i] = true$   
 Go to step 2.
4. If  $Finish[i] == true$  for all  $i$ , then the system is in a safe state.

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

### 7.5.3.2 Resource-Request Algorithm

Next, we describe the algorithm for determining whether 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:

1. If  $Request_i \leq Need_i$ , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If  $Request_i \leq Available$ , go to step 3. Otherwise,  $P_i$  must wait, since the resources are not available.

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

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

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

### 7.5.3.3 An Illustrative Example

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 ten instances, resource type  $B$  has five instances, and resource type  $C$  has seven instances. Suppose that, at time  $T_0$ , the following snapshot of the system has been taken:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
$P_0$	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  $\text{Max} - \text{Allocation}$  and is as follows:

	<u>Need</u>
	<i>A B C</i>
$P_0$	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  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  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  $\text{Request}_1 = (1, 0, 2)$ . To decide whether this request can be immediately granted, we first check that  $\text{Request}_1 \leq \text{Available}$ —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:

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  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 for students to implement the banker's algorithm.

## 7.6 Deadlock Detection

As we have noted, 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 may provide:

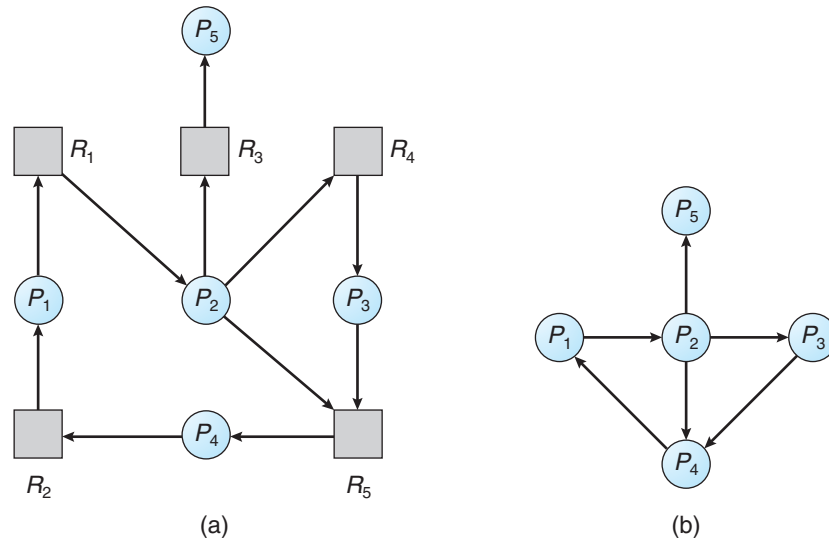
- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

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 7.11** (a) Resource-allocation graph. (b) Corresponding wait-for graph.

$R_q$ . For example, in Figure 7.11, 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.

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize *Work* = *Available*. For *i* = 0, 1, ..., *n*-1, if *Allocation<sub>i</sub>* ≠ 0, then *Finish*[*i*] = *false*; otherwise, *Finish*[*i*] = *true*.
2. Find an index *i* such that both
  - a. *Finish*[*i*] == *false*
  - b. *Request<sub>i</sub>* ≤ *Work*
 If no such *i* exists, go to step 4.
3. *Work* = *Work* + *Allocation<sub>i</sub>*  
*Finish*[*i*] = *true*  
 Go to step 2.
4. If *Finish*[*i*] == *false* for some *i*, 0 ≤ *i* < *n*, then the system is in a deadlocked state. Moreover, if *Finish*[*i*] == *false*, then process *P<sub>i</sub>* 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<sub>i</sub>* (in step 3) as soon as we determine that *Request<sub>i</sub>* ≤ *Work* (in step 2b). We know that *P<sub>i</sub>* is currently *not* involved in a deadlock (since *Request<sub>i</sub>* ≤ *Work*). Thus, we take an optimistic attitude and assume that *P<sub>i</sub>* 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<sub>0</sub>* through *P<sub>4</sub>* 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<sub>0</sub>*, we have the following resource-allocation state:

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
<i>P<sub>0</sub></i>	0	1	0	0	0	0	0	0	0
<i>P<sub>1</sub></i>	2	0	0	2	0	2			
<i>P<sub>2</sub></i>	3	0	3	0	0	0			
<i>P<sub>3</sub></i>	2	1	1	1	0	0			
<i>P<sub>4</sub></i>	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<sub>0</sub>*, *P<sub>2</sub>*, *P<sub>3</sub>*, *P<sub>1</sub>*, *P<sub>4</sub>*> 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:

	<u>Request</u>		
	A	B	C
$P_0$	0	0	0
$P_1$	2	0	2
$P_2$	0	0	1
$P_3$	1	0	0
$P_4$	0	0	2

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

### 7.6.3 Detection-Algorithm Usage

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

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

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

Deadlocks 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, then, 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, invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined 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, the resource graph may contain many cycles. In this case, we generally cannot 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 types 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 Summary

A deadlocked 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 deadlocked state.
- Allow the system to enter a deadlocked 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, rather than preventing them, requires that the operating system have a priori information about 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 each process may request. 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 may 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 some selected processes may never complete their designated tasks.

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.

## Practice Exercises

- 7.1 List three examples of deadlocks that are not related to a computer-system environment.
- 7.2 Suppose that a system is in an unsafe state. Show that it is possible for the processes to complete their execution without entering a deadlocked state.
- 7.3 A possible method for preventing deadlocks is to have a single, higher-order resource that must be requested before any other resource. For example, if multiple threads attempt to access the synchronization objects  $A \cdots E$ , deadlock is possible. (Such synchronization objects may include mutexes, semaphores, condition variables, and the like.) We can prevent the deadlock by adding a sixth object  $F$ . Whenever a thread wants to acquire the synchronization lock for any object  $A \cdots E$ , it must first acquire the lock for object  $F$ . This solution is known as **containment**: the locks for objects  $A \cdots E$  are contained within the lock for object  $F$ . Compare this scheme with the circular-wait scheme of Section 7.4.4.
- 7.4 Prove that the safety algorithm presented in Section 7.5.3 requires an order of  $m \times n^2$  operations.
- 7.5 Consider a computer system that runs 5,000 jobs per month and has no deadlock-prevention or deadlock-avoidance scheme. Deadlocks occur about twice per month, and the operator must terminate and rerun about