C H A P T E R  6

Deadlocks

6.1 DEADLOCK WITH REUSABLE AND CONSUMABLE RESOURCES
6.2 APPROACHES TO THE DEADLOCK PROBLEM
6.3 A SYSTEM MODEL
6.4 DEADLOCK DETECTION
6.5 RECOVERY FROM DEADLOCK
6.6 DYNAMIC DEADLOCK AVOIDANCE
6.7 DEADLOCK PREVENTION

We have already introduced the notion of deadlock informally with the dining philosophers problem in Section 3.3.2. In our solution, it is possible to reach a state where each philosopher acquires one fork and waits—indefinitely—for the other fork to become free. In general, whenever a process is blocked on a resource request that can never be satisfied because the resource is held by another blocked process, the processes are said to be deadlocked. This condition can only be resolved by an explicit intervention from the operating system (OS) or the user. Such intervention involves either the preemption of some of the resources held by the deadlocked processes, or the termination of one or more of the processes.

Why are deadlock studies important? It is clear that some action must be taken either to prevent deadlocks from occurring or to handle them whenever they do occur. If ignored, the deadlocked processes would exist indefinitely in a blocked state, never performing any useful work and wasting memory space and other resources, which cannot be used by other processes.

Since deadlocks are quite rare, yet difficult to manage efficiently, many general-purpose computing facilities simply ignore them. They are viewed as only an inconvenience, causing possible delays or interruption in service, and are handled when a user notices a system problem. However, deadlocks may become critical in a number of real-time applications, where any delay could result in loss of data or endanger human life. This includes, for example, real-time data communication, computer-aided manufacturing and process control, automated vehicle monitoring and control, and medical therapy and life-support systems in hospitals. Similarly, autonomous systems, such as those on board spacecraft, cannot always rely on timely human intervention. Thus, it is important that deadlocks be either prevented or resolved automatically.

This chapter contains a relatively formal treatment of the deadlock problem, based notably on the model of Holt (1971, 1972). Dijkstra (1968) was one of the first and most influential contributors in the area. More recently, the problem has been studied in the context of database and distributed systems. Following some examples and
definitions, deadlock detection, recovery, and prevention are discussed within a uniform graph model.

6.1 DEADLOCK WITH REUSABLE AND CONSUMABLE RESOURCES

Each resource element in the system can be identified with a given class, where the number of resource classes is fixed and depends on the particular system. For the purposes of deadlock modeling, it is sufficient to assume that individual units within each resource class are indistinguishable. For example, one resource class could be disk storage. Individual disk drives or even individual disk blocks represent the indistinguishable units within that resource class. We can divide all resource classes into two fundamentally different types: conventional nonshared objects and message-like objects, referred to as reusable and consumable resources, respectively.

6.1.1 Reusable and Consumable Resources

Reusable resources are permanent objects with the following properties:

1. The number of units within a class is constant; i.e., processes cannot create or delete units at runtime.
2. Each unit is either available, or is allocated to one and only one process; there is no sharing.
3. A process must first request and acquire a resource unit before it can release that unit.

The above definition captures the essential features of most conventional, nonshared, software and hardware resources that processes may use during their lifetimes. For example, hardware components, such as main memory, secondary storage, I/O devices, or processors, and software components, such as data files, tables, or semaphores, are considered reusable resource classes.

Consumable resources are produced and consumed dynamically, and have the following properties:

1. The number of units within a class varies at runtime; it may be zero and is potentially unbounded.
2. A process may increase the number of units of a resource class by releasing one or more units into that class; i.e., processes create new resources at runtime without acquiring them first from the class.
3. A process may decrease the number of units of a resource class by requesting and acquiring one or more units of that class. The units are not returned to the resource class but are consumed by the acquiring process.

Many types of data generated by either hardware or software have the above characteristics of consumable resources. The most prominent example are interprocess messages, which are generated by one process and consumed by another. Other examples include various types of synchronization and communication signals, interrupts, events, or data structures handed from one process to another. Although consumable resources are important, they are more difficult to manage than reusable resources, and there are fewer
formal results that may be used for deadlock detection or prevention. With the exception of an example in the next section, the remainder of this chapter covers exclusively reusable resources.

### 6.1.2 Deadlocks in Computer Systems

Deadlocks may occur with either reusable or consumable resources. The following two cases illustrate some differences.

**EXAMPLES: Deadlocks with Files and Messages**

1. **File Sharing.** Consider two processes \( p_1 \) and \( p_2 \), both of which must write to two files, \( f_1 \) and \( f_2 \). The files are considered reusable resources. Suppose that \( p_1 \) opens \( f_1 \) first and then \( f_2 \), and \( p_2 \) opens the files in the reverse order, as illustrated in the following code:

\[
\begin{align*}
\text{p1:} & \quad \text{p2:} \\
& \quad \vdots \quad \vdots \\
& \quad \text{open(f1, w);} \quad \text{open(f2, w);} \\
& \quad \text{open(f2, w);} \quad \text{open(f1, w);} \\
& \quad \vdots \quad \vdots
\end{align*}
\]

The two processes are assumed to execute concurrently, but we do not know anything about their relative speed. As long as \( p_1 \) opens both files before \( p_2 \) begins opening the first, or vice versa, there is no problem. Let’s say \( p_1 \) succeeds in opening both files. It will use and eventually close them. In the meantime, \( p_2 \) will be blocked on its first `open` statement, since the files must be opened in exclusive mode for writing (w). When \( p_1 \) closes the files, \( p_2 \) will reopen them and proceed unhindered.

The problem occurs when \( p_1 \) opens \( f_1 \) while, concurrently, \( p_2 \) opens \( f_2 \). This could happen easily in a time-sharing environment. For example, control could switch from \( p_1 \) to \( p_2 \) after \( p_1 \) opened \( f_1 \); \( p_2 \) would then open \( f_2 \). At this point, both processes are deadlocked. Each is holding one of the files open while trying to open the other. Since this will never be closed, the processes will wait indefinitely.

2. **Message-Passing.** Consider three processes, \( p_1 \), \( p_2 \), and \( p_3 \). Assume that the processes send messages (\( m \)) to each other along a ring, and that the `receive` operations are blocking. After an initial `send` from \( p_1 \) to \( p_2 \), each process repeatedly receives a message from its left-hand neighbor and sends a message to its right-hand neighbor. As long as the condition \( C \) is true and the initial `send` is performed, the above cycle continues indefinitely. However, if \( C \) is false, all three processes are blocked forever on their `receive` operations, waiting for a message that will never be sent.

\[
\begin{align*}
p1: & \quad p2: \quad p3: \\
& \quad \vdots \quad \vdots \quad \vdots \\
& \quad \text{if (C) send(p2,m);} \quad \text{while (1) {} \quad while (1) {} \quad while (1) {}} \\
& \quad \text{while (1) {}} \quad \text{while (1) {}} \quad \text{while (1) {}}
\end{align*}
\]
Deadlocks

A deadlock is a state where two or more processes are blocked indefinitely, waiting for each other. A process is considered blocked when it executes a synchronization or communication operation, such as \( P(s) \), or a blocking receive, and the requested resource (the semaphore \( s \) or the message) is currently not available. In Section 4.5.1, we described two ways to block a process within such an operation: busy-waiting or placing the process on a wait queue. In the first case, the process continues to run, whereas in the second case it is stopped. However, for the purposes of deadlocks both implementations are treated equally. A process is considered blocked until it leaves the synchronization or communication operation.

A condition related to deadlocks was already introduced in Section 2.3.1, where the third solution to the CS problem could result in indefinite postponement of both processes competing for the CS. This can occur if both processes proceed at exactly the same pace. Neither is blocked, yet they still make no real progress, since both keep asking for permission to enter the CS indefinitely. Such an “active” form of deadlock, commonly referred to as a livelock, leads to the starvation of both processes in the same way as an actual deadlock. Livelocks are similar to deadlocks where the blocking operation uses busy-waiting. In both cases, the processes consume CPU resources but make no progress.

However, deadlock and starvation are very different. Deadlock always leads to starvation of at least two processes, but starvation may have other causes. Processes may starve in a livelock fashion as above, or they can starve by being blocked for an unbounded amount of time, waiting for a resource that could be made available but never is. The latter case is not a deadlock situation. To illustrate these points, we present two memory allocation examples, one with deadlock and one with starvation.

Examples: Deadlock vs. Starvation

1. Deadlock on Memory Blocks

Suppose that two processes, \( p_1 \) and \( p_2 \), are competing for a memory resource containing four blocks. Let their code requests follow the sequences:

\[
\begin{align*}
p_1: & \quad \text{receive}(p_3, m); & \quad \text{receive}(p_1, m); & \quad \text{receive}(p_2, m); \\
& \quad \vdots & \quad \vdots & \quad \vdots \\
& \quad \text{send}(p_2, m); & \quad \text{send}(p_3, m); & \quad \text{send}(p_1, m); \\
& \quad \{ & \quad \{ & \quad \}
\end{align*}
\]

\[
\begin{align*}
p_2: & \quad \text{receive}(p_3, m); & \quad \text{receive}(p_1, m); & \quad \text{receive}(p_2, m); \\
& \quad \vdots & \quad \vdots & \quad \vdots \\
& \quad \text{send}(p_2, m); & \quad \text{send}(p_3, m); & \quad \text{send}(p_1, m); \\
& \quad \{ & \quad \{ & \quad \}
\end{align*}
\]
The function \textit{Get Mem}(n) requests \( n \) blocks of memory. Now, if \( p_1 \) and \( p_2 \) call the statements at labels \( a \) and \( c \) at the same time, three of the four memory units are allocated, leaving only one available unit. Deadlock occurs when the processes execute the subsequent statements at labels \( b \) and \( d \). Both processes become blocked, waiting for two units to become free. There is no legitimate sequence of operations that can break the deadlock; system or user intervention is necessary.

2. \textit{Starvation on Memory Blocks}

Real deadlocks involve at least two processes, but starvation (indefinite postponement) could happen to even a single process. Consider a system with 200 MB of main memory, and assume that any process requires either 100 MB or 200 MB to execute. Suppose further that two 100-MB processes are currently running, and that the ready queue always contains 100-MB processes. Then, whenever a process terminates, the system could always choose to load another 100-MB process, since that is how much memory is available. The 200-MB process will never get a chance to run, unless the scheduler postpones the loading of another 100-MB process. A similar situation occurs with schedulers that give highest priority to short processes; the scheduling of a long-running process could be postponed indefinitely. In both cases, the blocked process is not deadlocked, since the system does not need to starve them. For example, the system above could choose not to load the next 100-MB process and wait until 200 MB are available.

\section*{6.2 APPROACHES TO THE DEADLOCK PROBLEM}

Some systems choose to completely ignore deadlocks, assuming that possible deadlocks can always be resolved by explicit user interaction. For example, killing processes that do not seem to be making any progress or rebooting all or part of the entire system are typical responses. This “practical” approach usually involves \textit{timeouts} on waits for a resource. If a process is blocked for too long a time, a possible error, including deadlock, is indicated. It is then up to the programmer to provide code to handle each timeout; for example, by repeating the request at a later time or by taking a different course of actions.

In the many cases where the above strategies are not acceptable, we must implement mechanisms to explicitly handle the problem. The following options are possible:

1. \textbf{Detection and recovery}. Using this approach, we allow a deadlock to occur, but implement methods for detecting its presence and subsequently eliminating it.

2. \textbf{Avoidance}. This refers to dynamic schemes, where the system screens all resource requests. If granting a request would take the system into a state where deadlock could occur, the request is delayed until it become safe to grant it.

3. \textbf{Prevention}. This term is used for static techniques, where the rules governing the requests and acquisitions of resources are restricted in such a way that deadlock could never occur.

In the following sections, we present specific methods to implement the above three options for reusable resources.
6.3 A SYSTEM MODEL

We first present a graph model as a convenient notation for the process-resource state of a system, for the changes that occur as a result of resource requests, allocations, and releases, and for defining specific deadlock-related states. Our deadlock algorithms are directly based on the model.

6.3.1 Resource Graphs

To be able to reason about deadlocks, we must have a representation for processes, resources, and their relationships in the system. We will represent the state of the system by a directed graph, called the resource graph. All processes and resource classes are represented as vertices, and allocations and requests are represented as directed edges. Thus, at any given point in time, a resource graph captures all current allocations of resources to processes, and all current requests by processes for resources. The graph consists of the following components:

- **Processes.** Each process \( p_i \) (\( 1 \leq i \leq n \)) is a vertex represented as a circle. For example, the graph in Figure 6-1 shows a system consisting of three processes, \( p_1 \), \( p_2 \), and \( p_3 \).

- **Resources.** Each resource class \( R_j \) (\( 1 \leq j \leq m \)) is a vertex represented as a rectangle. Each individual (indistinguishable) resource unit within that class is represented as a small circle inside the rectangle. The sample graph in Figure 6-1 shows two resource classes, \( R_1 \) and \( R_2 \); \( R_1 \) consists of two units, and \( R_2 \) consists of three.

- **Resource requests.** A request by a process \( p_i \) for a unit of a resource \( R_j \) is represented as a directed edge (\( p_i \rightarrow R_j \)); this is called a request edge. Process \( p_1 \) of Figure 6-1 has no current requests, process \( p_2 \) is requesting two units of \( R_2 \), and process \( p_3 \) is requesting one unit of \( R_1 \).

- **Resource allocations.** An allocation of a resource unit of class \( R_j \) to a process \( p_i \) is represented as a directed edge (\( R_j \rightarrow p_i \)); this is called an allocation edge. Each allocation edge is connected to one of the circles within the class to indicate that the unit is currently unavailable to other processes. In Figure 6-1 process \( p_1 \) holds one unit of each resource class, \( p_2 \) holds one unit of \( R_1 \), and process \( p_3 \) holds two units of \( R_2 \).

![FIGURE 6-1. A resource graph.](image)
6.3.2 State Transitions

Each resource graph represents a particular state of the system. It captures all current allocations and pending requests by all processes. The system state changes to a new state whenever a process requests, acquires, or releases a resource. These are the only possible operations, since no other action can affect the system with respect to deadlock. The three operations are defined as follows:

1. **Request.** Any process \( p_i \) in a given state may request additional resource units of any resource class \( R_j \). The request operation changes the state of the system to a new state that contains the additional request edges \( (p_i \rightarrow R_j) \).

   Any request is subject to the following restrictions:
   - The requesting process \( p_i \) currently has no request edges from \( p_i \) to any resource classes. This condition represents the fact that a process is blocked from the time it issues a request until the request is granted. Consequently, the process cannot execute any operations during that time.
   - The number of edges between \( p_i \) and \( R_j \) (both request and allocation edges) must never exceed the total number of units in \( R_j \). Otherwise, the request could never be satisfied and the process would be blocked forever.

   Figure 6-2 illustrates the effect of a request operation. The leftmost resource graph shows a system consisting of two processes, \( p_1 \) and \( p_2 \), and a single resource class \( R \) with three resource units. In the system state \( S_0 \), process \( p_2 \) is holding one unit of \( R \); no requests are pending. The transition to the new state \( S_1 \) is caused by a request of \( p_1 \) for two units of \( R \). The state \( S_1 \) shows the corresponding request edges.

2. **Acquisition.** A process \( p_i \) in a given state may acquire previously requested resources. The acquisition operation changes the state of the system to a new state where the direction of all request edges \( (p_i \rightarrow R_j) \) of \( p_i \) is reversed to \( (R_j \rightarrow p_i) \) to reflect the allocations. Note that (unlike request and release) the

![FIGURE 6-2. State transitions by process operations.](image-url)
acquisition operation is not issued by the process \( p_i \) itself, but rather represent the request granting action taken by the resource manager. In fact, \( p_i \) is blocked from the time it issues a request until the acquisition is completed.

Any acquisition is subject to the following restriction:

- All outstanding requests of \( p_i \) must be satisfiable. That means, there must be a free resource (small circle inside \( R_j \)) for every request edge \((p_i \rightarrow R_j)\).

This guarantees that the process gets all its requested resources, or it remains blocked. The reason for disallowing partial allocations is that it simplifies the deadlock detection algorithms.

Consider again Figure 6.2. It shows that in state \( S_2 \), the process \( p_1 \) acquires the two previously requested units of \( R \); the two request edges in \( S_1 \) change to allocation edges in the new state \( S_2 \).

3. Release. A process \( p_i \) in a given state may release any previously acquired resource units. The release operation changes the state of the system to a new state where the allocation edges \((R_j \rightarrow p_i)\) corresponding to the released resource units are deleted. Any release is subject to the following restrictions:

- The process \( p_i \) currently has no request edges from \( p_i \) to any resource class. This is the same condition as in the case of the request operation.
- The process can release only those units it is currently holding; i.e., there must be an allocation edge \((R_j \rightarrow p_i)\) for any resource units being released.

The last part of Figure 6.2 illustrates the effect of a release operation. Process \( p_1 \) releases one of the previously acquired units of \( R \).

6.3.3 Deadlock States and Safe States

Consider again the three state transitions shown in Figure 6.2. Note that these do not represent the only possible sequence of state changes for this system. The set of possible states the system can enter depends on the operations the processes can perform in a given state. For example, in state \( S_0 \) of Figure 6.2, \( p_2 \) could release the unit of \( R \) it is holding, leading to a new state with no edges. Alternately, \( p_1 \) could choose to request only one unit of \( R \), or possibly all three; each choice would lead to a new system state. We do not generally know which path a given process may take through its code. We also do not know the order in which processes will be interleaved. But there is a finite set of states the system may potentially enter. Our ultimate goal is to identify (or prevent) those states containing deadlocked processes. We first define the following key terms:

- A process is **blocked** in a given state if it cannot cause a transition to a new state; i.e., the process can neither request, acquire, or release any resources in that state, because some of the restrictions imposed on these operations are not satisfied. Assume, for example, that process \( p_1 \) in state \( S_0 \) of Figure 6.2 requests three units of \( R \) instead of just two. It would become blocked in the new state until \( p_2 \) released the one unit it is currently holding.

- A process is **deadlocked** in a given state \( S \) if it is blocked in \( S \) and if no matter what operations (state changes) occur in the future, the process remains blocked. Assume, for example, that in the state \( S_3 \) of Figure 6.2, process \( p_1 \) requests two
more units or $R$, and in the next transition, process $p_2$ also requests two more units or $R$. Both processes would become blocked and remain blocked forever, i.e., deadlocked.

- A state is called a **deadlock state** if it contains a deadlocked process. Note that a cycle in the resource graph is a necessary condition for deadlock.

- A state $S$ is a **safe state** if for all states $S'$ that can be reached from $S$ using any sequence of valid request, acquire, and release operations, $S'$ is not a deadlock state.

### EXAMPLE: Reachable States of a System

Consider again the example of Section 6.1.2 where two processes $p_1$ and $p_2$ each must write to two files. Let us represent these files as two resources classes, $R_1$ and $R_2$, each containing one resource unit. Assume that $p_1$ always requests $R_1$ before $R_2$, and it releases $R_2$ before $R_1$. The sequence for $p_2$ is analogous, except that it requests and releases the resources in the reverse order.

Figure 6-3 illustrates the system state transitions if we consider only $p_1$ in isolation. The top portion shows the individual resource graphs, whereas the bottom portion shows possible state changes. In $S_0$, the resource graph has no edges; in $S_1$, $p_1$ is requesting $R_1$; in $S_2$, $p_1$ has acquired $R_1$; at this point it can release $R_1$, which takes it to back to $S_0$, or it can request $R_2$, which takes it to $S_3$; when it acquires $R_2$, the system is in $S_4$, where $p_1$ is holding both resources; releasing $R_2$ takes it back to $S_2$, and so on.

The state transitions of $p_2$, when considered in isolation, result in an analogous state transition diagram. If we now combine the processes, assuming that both can be running concurrently, we obtain the two-dimensional state transition diagram for all reachable states of the system, as shown in Figure 6-4. Each horizontal transition from a state $S_{i,j}$ to $S_{i+1,j}$ represents an operation by $p_1$. Each vertical transition from a state $S_{i,j}$ to $S_{i,j+1}$ represents an operation by $p_2$. For example, in $S_{1,1}$, each process has a request edge for one of the resources; in $S_{2,2}$, each process has an allocation edge for one of the resources; in $S_{3,3}$, each process has both an allocation edge for one of the resources and a request edge for the other resource. Figure 6-4 shows the resource graph for the state $S_{3,3}$, which is a deadlock state.

![Figure 6-3](image-url)

**FIGURE 6-3.** State transitions caused by process $p_1$. 
FIGURE 6-4. State transitions caused by $p_1$ and $p_2$.

The state transition diagram in Figure 6-4 also illustrates the other concepts introduced above:

- $p_1$ is blocked (but not deadlocked) in $S_{3,2}$ and $S_{1,4}$; there is no horizontal transition leading from these states. Similarly, $p_2$ is blocked (but not deadlocked) in $S_{2,3}$ and $S_{4,1}$.

- Both processes are deadlocked in $S_{3,3}$, since there is no transition, horizontal or vertical, leading from this state; i.e., $S_{3,3}$ is a deadlock state.

- No state is safe, since the deadlock state $S_{3,3}$ is reachable from any other state.

### 6.4 DEADLOCK DETECTION

To detect whether a given state $S$ is a deadlock state, it is necessary to determine whether the processes that are blocked in $S$ will remain blocked forever. This can be accomplished using a technique called **graph reduction**, which mimics the following execution scheme:
First, all requests satisfiable in state $S$ for unblocked processes are granted, and the requesting processes continue to completion without requesting any further resources. Prior to termination, they release all their resources. These actions may wake up previously blocked processes, which then proceed to completion in the same manner. This is repeated until there are either no processes left, i.e., all processes have been terminated, or all remaining processes are blocked. In the latter case, the original state $S$ is a deadlock state.

We now consider the graph reduction algorithm in more detail.

### 6.4.1 Reduction of Resource Graphs

Given a resource graph representing a system state, repeat the following steps until there are no unblocked processes remaining:

1. Select an unblocked process $p$.
2. Remove $p$, including all its request and allocation edges.

A resource graph is called **completely reducible** if, at the termination of the above reduction sequence, all process nodes have been deleted. We can now obtain the following important results in deadlock detection:

- $S$ is a deadlock state if and only if the resource graph of $S$ is not completely reducible.
- All reduction sequences of a given resource graph lead to the same final graph.

That means, in step 1 of the reduction sequence above, it does not matter which unblocked process is selected.

The two results lead to an efficient algorithm for deadlock detection. They imply that, given a resource graph for a state $S$, we can apply any sequence of reductions. If, at the end, the graph is not completely reduced, $S$ is a deadlock state.

**EXAMPLE: Graph Reduction**

To illustrate the graph reduction algorithm, consider the resource graph in Figure 6-5a. The only unblocked process in this state is $p_1$. Figure 6-5b shows the graph after reducing it by $p_1$. This unblocks $p_2$ and $p_4$, which can be removed in either order. The resulting state, shown in Figure 6-5c is irreducible, since both $p_3$ and $p_5$ are blocked. Hence, the original state was a deadlock state.

### 6.4.2 Special Cases of Deadlock Detection

The general results of the last section can be used to produce more efficient detection algorithms by taking advantage of a number of practical restrictions.

**Testing for a Specific Process**

On some occasions, we may be interested in determining only whether a specific process $p$ is deadlocked, rather than testing the entire system state. This can be achieved by applying the graph reduction algorithm until one of the following occurs: 1) the graph
can be reduced by $p$, which implies that $p$ is not blocked and hence not deadlocked; and 2) the graph is irreducible, in which case $p$ is deadlocked. In both cases, the algorithm stops at this point.

**Continuous Deadlock Detection**

Testing for deadlock can be accomplished more efficiently if it is done on a continuous basis. If we know that the current state $S$ is not deadlocked, then the next state $S'$ is a deadlock state if and only if the operation that caused the transition was a request, and the process $p$ that performed the operation is deadlocked in $S'$. In other words, a deadlock in $S'$ can only be caused by a request that cannot be granted immediately. Thus, we only must check the specific process $p$; if $p$ can be reduced, $S'$ is not a deadlock state.

**Immediate Allocations**

Some resource allocators adopt the simple policy that all satisfiable requests are always granted immediately. If this is the case, a resource graph never contains any satisfiable request edges, since these are immediately turned into allocation edges. Such system states are referred to as **expedient**. For example, the state of Figure 6-5a is not expedient because $p_1$ is not blocked, and its request can be granted immediately. If we allocate a unit of $R_1$ to $p_1$, the state then becomes expedient. This yields a simpler condition for deadlock detection:

*If a system state is expedient, then a knot in the corresponding resource graph implies a deadlock.*

A **knot** in a directed graph is defined as a subset of nodes that satisfy the following two conditions: 1) Every node within the knot is reachable from every other node within
the knot; and 2) a node outside the knot is not reachable from any node inside the knot. For example, consider Figure 6-5a again. If we delete the edge $R_2 \rightarrow p_1$, the graph has a knot \{ $R_2$, $p_3$, $R_3$, $p_5$ \}. If we also change the edge $p_1 \rightarrow R_1$ to $R_1 \rightarrow p_1$, the state becomes expedient, and it contains a deadlock.

To understand the intuition behind the above deadlock-detection condition, consider a resource graph containing a knot $K$. All processes within $K$ must have pending requests, because each node must have an outgoing edge. Furthermore, they may only be requesting resources within $K$, since no edges may lead outside of $K$. If the state is expedient, none of the pending requests can be satisfiable, and hence all the processes must be deadlocked. Note that a knot is sufficient for deadlock, but not a necessary condition. The expedient state that we produced from Figure 6-5a by reversing the edge $p_1 \rightarrow R_1$, for example, does not have a knot but is still deadlocked.

**Single-Unit Resources**

In many situations, all resource classes are limited to having only a single unit. Common examples are files and locks. With this restriction, the existence of a simple cycle in the resource graph implies a deadlock. Thus, in this case, a cycle is both a necessary and sufficient condition for deadlock.

To show why this condition is sufficient, assume that the graph contains a cycle $C$. Since every process within $C$ must have an entering and exiting edge, it must have an outstanding request for a resource in $C$ and must hold resources in $C$. Therefore, every process in $C$ is blocked on a resource in $C$ that can be made available only by another process in $C$. Hence, all the processes in the cycle are deadlocked.

Thus, to detect a deadlock in systems with single-unit resources, we only must test the resource graph for cycles. There are well-known algorithms that can accomplish that in $O(n^2)$ time, where $n$ is the number of nodes.

A single-unit resource class can have only a single allocation edge attached to it. This allows us to present the resource graph in a simplified form. We omit all resource classes and, instead, point all request edges directly at the process currently holding the resource. Such a graph is called a wait-for graph because an edge $(p_i \rightarrow p_j)$ indicates that process $p_i$ is blocked on a resource currently held by $p_j$; i.e., $p_i$ is waiting for $p_j$ to release the resource. Then, more simply, a cycle in the wait-for graph is a necessary and sufficient condition for deadlock.

**EXAMPLE: Wait-For Graph**

Figure 6-6 illustrates the concept. In Figure 6-6a, processes $p_1$ and $p_2$ are requesting a resource currently held by $p_3$. In turn, $p_3$ is requesting a resource currently held by $p_4$. Figure 6-6b shows the corresponding wait-for graph: $p_1$ and $p_2$ are waiting for $p_3$, which is waiting for $p_4$.

**6.4.3 Deadlock Detection in Distributed Systems**

Wait-for graphs as defined in the previous section are the basis of deadlock detection in distributed systems restricted to single-unit resources. The added difficulty is that no single machine has a complete picture of all resource requests or allocations. Thus, the
wait-for graph itself is distributed; some edges cross the boundary between different machines.

Let each machine have its own local coordinator, responsible for maintaining the local portion of the wait-for graph. To detect deadlock, we must check for cycles in the global wait-for graph. But since cycles can span multiple machines, the coordinators must cooperate by exchanging information with one another to detect such global cycles. This can be done in two different ways, either through a central coordinator or in a distributed approach.

**Central Coordinator Approach**

The simplest way to detect cycles in the global wait-for graph is to mimic a centralized system. One of the local coordinators is designated as the central coordinator. It collects the local wait-for graphs from all other coordinators, assembles them into a complete global graph, and analyzes the global graph for the presence of cycles. The local graphs can be sent to the global coordinator whenever an edge is added or removed, or (less frequently) by grouping together multiple changes to reduce message traffic.

The centralized approach to deadlock detection, although straightforward to implement, has two main drawbacks. First, the global coordinator becomes a performance bottleneck and a single point of failure. Second, it is prone to detecting nonexisting deadlocks, referred to as phantom deadlocks. To illustrate how this can happen, consider a system with two machines, $M_1$ and $M_2$, each holding two processes. Figure 6-7a shows the global wait-for graph. This indicates that $p_1$ is currently holding a resource that $p_2$ is requesting; $p_2$, in turn, is holding a resource that $p_3$ (on machine $M_2$) is requesting; and $p_3$ is holding a resource that $p_4$ is requesting. Assume now that $p_1$ requests a resource held by $p_4$, which adds the edge ($p_1 \rightarrow p_4$). Concurrently, $p_3$ on $M_2$ times out on its wait, which removes its waiting edge for $p_2$, i.e., ($p_3 \rightarrow p_2$). If the global coordinator receives the update from $M_1$ first, it will construct the graph shown in Figure 6-7b, and, consequently, it will report a deadlock. In reality, no deadlock exists, since the actual wait-for graph does not contain the phantom edge ($p_3 \rightarrow p_2$).
Section 6.4  Deadlock Detection  191

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig6-7}
\caption{A phantom deadlock.}
\end{figure}

**Distributed Approach**

A fully distributed deadlock-detection algorithm attempts to find cycles in the global wait-for graph by tracing the different paths through the graph without gathering it in one central location. The basic idea is to send a special message, called a probe, which replicates itself along all outgoing edges. If one of the replicas reaches the original destination of the probe, a cycle, and thus a deadlock, has been detected. There are many different implementations of this basic concept, referred to as edge-chasing or path-pushing approaches. What distinguishes them is the type and amount of information carried by the probe, and the type and amount of information that must be maintained by each local coordinator.

Let us consider the conceptually simplest approach, where each probe is the concatenation of edges it has traversed so far. The initial probe is launched by a process \( p_i \) when it becomes blocked on a request for a resource currently held by another process \( p_j \). The local coordinator extends the probe by adding to it the edge \((p_i \rightarrow p_j)\) and replicates it along all edges emanating from \( p_j \). If there are no such edges, indicating that the current process is not blocked, the probe is discarded. A cycle is detected whenever the process appears on the probe twice, indicating that the probe must have returned to an already-visited process.

**EXAMPLE: A Probe**

Figure 6-8 illustrates the above algorithm using seven processes distributed over four different machines. Assume that \( p_1 \) requests a resource held by \( p_2 \). Process \( p_1 \) creates the...
Chapter 6  Deadlocks

FIGURE 6-8. A probe to detect cycles.

initial probe containing the edge $p_1 \rightarrow p_2$. Since $p_2$ waits for $p_3$, the probe is extended to $p_1 \rightarrow p_2 \rightarrow p_3$. $p_3$ is waiting for $p_4$ and $p_7$, and consequently, replicates the probe along the two different paths, as shown. The one carrying $p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_5 \rightarrow p_1$ is discarded on machine $M_3$, since $p_7$ is not blocked. On the other hand, the other probe continues through $p_4 \rightarrow p_5$. At $p_5$, the probe is again replicated. The one containing $p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \rightarrow p_5 \rightarrow p_1$ closes the cycle upon reaching machine $M_1$. $p_1$ appears twice on the path—once as the initiator of the original probe and a second time as its recipient. Thus, the coordinator on $M_1$ detects a deadlock. In a similar manner, the coordinator on $M_2$ can detect the other cycle in the graph by noticing that $p_4$ appears twice on the probe $p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \rightarrow p_5 \rightarrow p_1 \rightarrow p_4 \rightarrow p_4$.

The main drawback of the above implementation is that the probe’s length is unbounded. That may not be a serious problem in practice, since the paths in the wait-for graphs are typically short. Nevertheless, there are approaches where the probe length can be kept constant at the expense of maintaining additional information about the already-known dependencies by each local coordinator (Chandy, Misra, Haas 1983).

6.5 RECOVERY FROM DEADLOCK

A deadlock always involves a cycle of alternating process and resource nodes in the resource graph. The two general approaches to recovery are process termination and resource preemption. In the first case, nodes and edges of the resource graph are eliminated. The second strategy involves edge deletions only. In both cases, the goal is to break the deadlock cycle.

6.5.1 Process Termination

The simplest and crudest recovery algorithms terminate all processes involved in the deadlock. This approach is unnecessarily wasteful, since, in most cases, eliminating a single process is sufficient to break the deadlock. Thus, it is better to terminate processes one at a time, release their resources, and at each step check if the deadlock still persists.
This is repeated until the deadlock is eliminated, or, in the worst case, until all but one of the originally deadlocked processes must be liquidated.

With this incremental method, we must decide on the order in which the processes will be terminated. Picking the processes at random is the easiest solution, but it is more rational to consider the cost of terminating different processes. Termination cost may involve a combination of the following:

1. **The priority of the process.** This metric is similar to that used for process scheduling. It may involve the process type (e.g., real-time, interactive, or batch), its CPU or memory requirements, and other metrics.

2. **The cost of restarting the process.** Many processes maintain very little state and can easily be restarted. This includes most interactive processes, such as the user shell, Internet browsers, or even text editors (provided the edited files are being saved periodically). In contrast, other applications cannot be “resumed” but must be repeated from the beginning. This includes, for example, scientific computations, which may run for hours or even days in batch mode. Terminating such applications is not only costly but also frustrating for the users.

3. **The current state of the process.** Many processes cooperate with others in a variety of ways, thus killing one process may seriously impact others that depend on it. For example, killing the consumer or the producer process in a producer-consumer scenario will leave the other process hanging. Similarly, killing a process in the middle of a CS will deadlock other processes that compete for the same CS. Finally, some operations are not idempotent, i.e., cannot be repeated without side effects. For example, appending data to a file cannot simply be repeated by rerunning a killed process.

In general, choosing the best sequence of processes to terminate is highly dependent on the particular system and its applications.

### 6.5.2 Resource Preemption

Resource preemption means taking away the contested resources from one or more of the deadlocked processes. This can be done in one of two ways. First, some resources may lend themselves to **direct preemption**; i.e., the system temporarily deallocates the resource, lets other processes use it, and gives it back to the original process. Few resources can be handled in this way transparently. For example, temporarily reallocating a printer in the middle of a print job would result in interleaved pages from multiple processes. Main memory is one of the few resources that can be preempted transparently, by temporarily swapping a process or some of its data to disk, and reloading it later when memory is again available.

An *indirect* form of resource preemption is achieved by **process rollback**. Some systems take periodic snapshots of processes, called **checkpoints**, to achieve fault tolerance. In the case of a crash, the process does not need to be restarted from its beginning. Instead, the process can be rolled back by restoring its last checkpoint and resuming its execution from there. We can use the same mechanism for resource preemption. To take away a resource from a process, we can roll it back to a checkpoint during which it had not acquired this resource yet. After resuming execution from this point, the process
would repeat the earlier request for the resource, which, in the meantime, could be used by other processes.

6.6 DYNAMIC DEADLOCK AVOIDANCE

The previous section focused on detecting deadlocks and eliminating them once they occur. An entirely different strategy is to prevent deadlocks from developing in the first place. When this is done through rules checked and enforced at runtime, the approach is referred to as **deadlock avoidance**.

6.6.1 Claim Graphs

The basic principle of deadlock avoidance is to delay the acquisition of resources that might cause the system to enter a deadlock state in the future. This can be determined if information about future process resource needs is available. Generally, processes do not know which resources they will need in the future. However, they can specify an upper bound on their needs, called the **maximum claim**. This is the largest number of units of each resource that the process will ever need at any one time.

The maximum claim of a process can be represented as a **claim graph**, which is an extension of the general resource graph used for deadlock detection. A claim graph, in addition to showing processes, resources, request edges, and allocation edges, contains a set of **potential request edges**. In the initial system state, the number of such edges \((p_i \rightarrow R_j)\) represent the maximum number of units of resource \(R_j\) that process \(p_i\) will ever need.

Each potential request edge may, in the future, be transformed into an actual request edge and, subsequently, into an assignment edge. But the total number of edges between the process \(p_i\) and resource class \(R_j\) (i.e., the sum of request, allocation, and claim edges) will always remain the same.

**EXAMPLE: Claim Graph**

Figure 6-9 shows an example of a claim graph in the initial system state \(S_0\), and its transformation into other states as the result of operations by the two processes. The request and allocation edges are shown as before; the added claim edges are represented by dashed lines.

The graph shows that process \(p_1\) may request, at most, two units of \(R\) at any given time, whereas \(p_2\) may request all three. The first operation is a request by \(p_2\) for one unit of \(R\). The new state \(S_1\) shows one of the claim edges transformed into an actual request edge. The request is granted, resulting in state \(S_2\). The next operation is a request by \(p_2\) for two units of \(R\) (in state \(S_3\), which is also granted (in state \(S_4\)). At this point, \(p_2\) could request one of two more units of \(R\), which would block the process until the units were released by \(p_1\).

6.6.2 The Banker’s Algorithm

Claim graphs may be used for dynamic deadlock avoidance. This is achieved by disallowing any **acquisition** operations unless the resulting claim graph is completely reducible. The fact that the claim graph can be completely reduced means that the worst case of all
possible future requests could be handled. That means, even if all processes requested the remainders of their claims, all requests could still be satisfied. The following theorem has been proven formally:

*If acquisition operations that do not result in a completely reducible claim graph are prohibited, any system state is safe.*

This theorem is the basis of deadlock-avoidance algorithms, the best known of which is the **banker’s algorithm** (Dijkstra 1968). It takes its name from an analogy to banking systems, where resource classes correspond to currencies, allocations correspond to loans, and maximum claims are considered the credit limits. Deadlock prevention is accomplished by the “banker,” which grants or delays any request in a given state $S$ as follows:

1. Assume the request in state $S$ is granted, i.e., temporarily change the request edge(s) into the corresponding acquisition edge(s); the new tentative state is $S'$. 
2. Reduce the claim graph of $S'$. That means, treat all claim edges in $S'$ as actual request edges and reduce the resulting graph. 
3. If the graph is completely reducible, grant the original request, i.e., accept $S'$ as the new state and continue; otherwise, delay the acquisition, i.e., revert to the original state $S$ and keep the request as pending.

**EXAMPLE:** **The Banker’s Algorithm**

To illustrate the working of the banker’s algorithm, consider the claim graph in Figure 6-10a, which shows the state of the same system as Figure 6-1, but at an earlier time. The requests by $p_1$ and $p_2$ for one unit of $R_1$ each have not been granted yet. Thus, all three processes have pending requests for one unit of $R_1$. The original resource graph also has been augmented by the claim edges of each process, shown as dashed lines. The task of the banker’s algorithm is to decide which of the three pending requests for $R_1$ can safely be granted.
Chapter 6  Deadlocks

![Diagram](image)

**FIGURE 6-10.** Deadlock avoidance using the banker’s algorithm.

Figure 6-10b shows the new graph should $p_1$’s request be granted. The resulting graph is completely reducible (first by $p_1$, then by $p_3$, and finally by $p_2$). Hence, this request could be granted in this state. The request by $p_3$ also results in a completely reducible graph and may be granted. On the other hand, the request by $p_2$ results in the graph shown in Figure 6-10c. If we interpret all claim edges as actual request edges, then all three processes are blocked, and the graph is not reducible. To prevent such a situation from developing, the request by $p_2$ must be delayed until it becomes safe to grant it.

**Special Case: Single-Unit Resources**

When all resource classes contain only a single unit, the reducibility of the claim graph can be tested much more efficiently. This is based on the following observation: a claim graph becomes irreducible only when an acquisition creates a cycle in the claim graph. Thus, a simple path-tracing algorithm from the acquiring process can be employed for deadlock prevention.
EXAMPLE: Single-Unit Resources

Figure 6-11a illustrates this idea using a simple system of two processes and two resources. The pending request by \( p_2 \) for \( R_1 \) must not be granted, since doing so would close the (directed) cycle \( p_1 \rightarrow R_1 \rightarrow p_2 \rightarrow R_2 \rightarrow p_1 \).

![Diagram](image)

**FIGURE 6-11.** Deadlock avoidance with single-unit resources.

We also can easily test for safeness of states in single-unit resource graphs. This is based on the following observation: a directed cycle can develop only if the graph contains an undirected cycle. In other words, if we disregard the direction of all edges and determine that the graph contains no undirected cycles, then no directed cycle will develop in such a claim graph. Consequently, in this situation, all states are safe, and there is no possibility of deadlock. Note that this only must be checked once, when the claim graph is first created. Thus, dynamic deadlock avoidance turns into a single static test.

EXAMPLE: Safe States

Figure 6-11b illustrates a system where all states are safe. If we ignore the direction of the claim edges, we see that no cycle can be formed, since the claim edges are a superset of all possible future request or acquisition edges.

6.7 DEADLOCK PREVENTION

Dynamic deadlock avoidance relies on *runtime checks* to ensure the system never enters an unsafe state. These are performed by the system (the resource manager) that is responsible for granting or delaying any given request. On the other hand, deadlock prevention relies on imposing additional rules or conditions on the system so that all states are safe. Thus, prevention is a *static* approach, where all processes must follow certain rules for requesting or releasing resources.

By examining the general form of a resource graph, we can identify the following three structural conditions that must hold for deadlock to occur:

1. **Mutual exclusion.** Resources are not sharable; i.e., there is, at most, one allocation edge from any resource unit to a process in the resource graph.
2. **Hold and wait.** A process must be holding a resource and requesting another; i.e., there must be an allocation edge from a resource unit to a process $p_1$, and a request edge from $p_1$ to another resource unit. The two units can be within the same resource class or within different classes.

3. **Circular wait.** At least two processes must be blocked waiting for each other; i.e., the graph must contain a cycle involving at least two processes and at least two resource units, such that each process holds one of the units and is requesting another.

The elimination of any of these three conditions would make it structurally impossible for deadlock to occur.

### 6.7.1 Eliminating the Mutual-Exclusion Condition

If all resources could be shared (accessed concurrently), no process would block on a resource, and no deadlock would occur. Some resources that might be used nonexclusively in a shared fashion are pure program code, read-only data files or databases, and clocks. Unfortunately, mutual exclusion is a fundamental requirement for correct use of many resource types, and this condition cannot generally be eliminated. For example, files cannot usually be written into by multiple processes at the same time in a meaningful way. Similarly, database transactions require exclusive access to records to ensure data consistency.

In some instances, it is possible to circumvent the mutual exclusion requirement by transforming nonsharable resources into sharable ones. The prime example is spooling of output, which was briefly discussed in the Introduction. Printers or other output devices always must be accessed by only one process at a time to prevent interleaving of output. To avoid unnecessary blocking of processes waiting for output, many OSs provide virtual devices, implemented as software files, into which processes may direct their output. When the complete output sequence is available, it is sent to the actual hardware printer, and the process continues without blocking.

### 6.7.2 Eliminating the Hold-and-Wait Condition

The simplest way to eliminate this condition is to insist that every process requests all resources it will ever need at the same time. A process with resources already allocated to it will never be blocked because it cannot make any further requests. It will eventually release all its resources (not necessarily at the same time), which may unblock other processes. But a given process will have either assignment edges or request edges, but never both together. Deadlock is therefore impossible, and every state is safe.

The main drawback of this simple policy is poor resource utilization—resources must be allocated well ahead of their actual use and be unavailable to other processes for possibly long periods of time. Resources also may be requested unnecessarily in anticipation of a use that does not materialize.

A more flexible approach is allow processes to request resources as they need them, but to always release all resources they are currently holding, prior to making any new request. This may result in having to repeatedly request and release frequently used resources. Assume, for example, that a process needs either resources $R_1$ and $R_2$, or
Section 6.7  Deadlock Prevention

$R_1$ and $R_3$, but never $R_2$ and $R_3$ together. To eliminate the hold-and-wait condition, the processes cannot simply release and request $R_2$ or $R_3$ as needed, while keeping $R_1$. Instead, the process must release both $R_1$ and $R_2$, prior to requesting $R_1$ and $R_3$, and vise versa.

Yet a third alternative is to give each process the ability to test whether a needed resource $R$ is currently available. If not, the process must release all other resources it is currently holding, prior to waiting for the unavailable resource $R$ to become free. None of these alternatives is really satisfactory.

6.7.3 Eliminating the Circular-Wait Condition

A cycle in the resource graph can develop only when processes request the same resources but in a different order. This can be prevented if all processes request all resources in the same order. Since it is not known, generally, which resources will be needed by which process, we assign a sequential ordering, $SEQ$, to all existing resource classes, such that $SEQ(R_i) \neq SEQ(R_j)$ for all $i \neq j$. In other words, any process can test whether a given resource precedes or succeeds another resource according to the ordering $SEQ$. Processes are then allowed to request any subset of the existing resources as in the general case, but they must request them in the order prescribed by $SEQ$. When a process already holds a resource $R_i$, it may only request resources $R_j$ with a higher sequence number, i.e., resources where $SEQ(R_i) < SEQ(R_j)$.

**EXAMPLE: Ordered Resources Policy**

Assume a system with four resource classes, $R_1$, $R_2$, $R_3$, and $R_4$, where $SEQ(R_1) < SEQ(R_2) < SEQ(R_3) < SEQ(R_4)$. Figure 6-12 shows three processes competing for these three resource classes. Process $p_1$ has no outstanding requests and could request units from any of the four classes. Process $p_2$ is already holding a unit from the highest class, $R_4$, and is not allowed to make further requests. Process $p_3$ holds units from $R_1$, $R_2$, and $R_3$; thus, the only additional resources it could request are from class $R_4$.

By assigning the most expensive or scarce resources to the highest classes, the requests of the most valuable resources can be deferred until they are actually needed. Nevertheless, the overall resource utilization of this ordered resources policy is still not very high, because some resources must be allocated well in advance of their need. This
policy was originally devised by Havender (1968) for the IBM OS called OS/360. It also is used to prevent deadlock on monitor locks when using nested monitors, i.e., when a procedure within one monitor calls a procedure within another monitor. These monitors must be called in a given order (Lampson and Redell 1980).

CONCEPTS, TERMS, AND ABBREVIATIONS

The following concepts have been introduced in this chapter. Test yourself by defining and discussing each keyword or phrase.

| Banker’s algorithm | Knot |
| Central coordinator | Livelock |
| Checkpoint | Maximum claim |
| Circular wait | Ordered resource policy |
| Claim graph | Phantom deadlock |
| Consumable resource | Probe message |
| Continuous detection | Reachable state |
| Deadlock | Reducible graph |
| Deadlock avoidance | Resource graph |
| Deadlock detection | Reusable resource |
| Deadlock prevention | Rollback |
| Deadlock recovery | Safe state |
| Deadlock state | Single-unit resource |
| Distributed detection | Starvation |
| Graph reduction | State of a system |
| Hold and wait | Wait-for graphs |

EXERCISES

1. Consider the dining philosophers problem of Section 3.2. Assume there are three philosophers, \( p_1, p_2, \) and \( p_3 \) using three forks, \( f_1, f_2, \) and \( f_3. \) The philosophers execute the following code:

```c
p1() {
    while (1) {
        P(f1); P(f1); P(f3);
        P(f3); P(f2); P(f2);
        eat; eat; eat;
        V(f3); V(f2); V(f2);
        V(f1) }
}
```

(a) Is deadlock possible in this system?
(b) Would deadlock be possible if we reversed the order of the P operations in process \( p_1, p_2, \) or \( p_3? \)
(c) Would deadlock be possible if we reversed the order of the V operations in process \( p_1, p_2, \) or \( p_3? \)
2. Consider the following reusable resource graph:

(a) Which processes are blocked?
(b) Which processes are deadlocked?
(c) Is the state a deadlock state?
(d) Does the graph contain a knot? If so, list the nodes constituting the knot.

3. Consider three processes, \(p_1\), \(p_2\), and \(p_3\), executing asynchronously the following sequences of code:

\[
\begin{align*}
p_1 & : \quad \ldots \quad P(x) \quad \ldots \quad V(x) \quad \ldots \\
p_2 & : \quad \ldots \quad P(y) \quad \ldots \quad V(z) \quad \ldots \\
p_3 & : \quad \ldots \quad P(z) \quad \ldots \quad V(z) \quad \ldots \\
\end{align*}
\]

The arrow in each line indicates which instruction the corresponding process is currently executing. All semaphores were initially set to 1.

(a) Draw a reusable resource graph describing the above situation where each semaphore is interpreted as a resource, and \(P\) and \(V\) operations represent requests and releases of the resources.

(b) Reduce the graph as much as possible; does it represent a deadlock state?

(c) If you could increase the number of units of any of the three resources, which increase (if any) would resolve the deadlock?

4. Using the definitions of Section 6.3.3, show that safeness of a state and deadlock are not complementary; i.e., the statement “\(S\) is not a deadlock state” does not imply that “\(S\) is a safe state”.

5. The basic graph-reduction algorithm given in Section 6.4.1 is very inefficient. The first step—finding an unblocked process during each iteration—requires a number of operations proportional to the number of processes. Develop a more efficient algorithm that maintains a list of unblocked processes. At each iteration, a process is selected from this list and removed from the graph. This reduction may, in turn, unblock other processes, which are added to the list. Thus, finding an unblocked process is always a single constant operation. (Hint: Maintain a counter for each process that records the number of resource classes the process is blocked on.)
202 Chapter 6  Deadlocks

6. Modify the state transition diagram of Figure 6-4 to reflect the following changes:
   (a) A process may release both resources at the same time.
   (b) A process will not release its first resource until it has acquired the second. Both
       resources are then released at the same time.
   (c) Process $p_1$ only needs resource $R_2$.
   (d) Would the changes in (a), (b), or (c) eliminate the possibility of deadlock?

7. Two processes, $p_1$ and $p_2$, both need two single-unit resources, $R_1$ and $R_2$. The
   processes repeatedly execute the following sequences (at unknown speeds):
   
   ```
   p1: while(1) {
       request R1; request R2;
       request R2; request R1;
       release R1; release R2;
       release R2; release R1;
       ...
   }
   p2: while(1) {
       request R2; request R1;
       request R1; request R2;
       release R1; release R2;
       ...
   }
   ```
   
   (a) Assume first that process $p_2$ is not executing. Draw the state transition diagram,
       similar to Figure 6-3, where each state $S_i$ corresponds to the state $i$ of $p_1$. In
       which state(s), if any, is $p_1$ blocked and/or deadlocked?
   (b) Now draw the state transition diagram for both processes, similar to Figure 6-4.
       Each system state $S_{i,j}$ represents the state $i$ of $p_1$ and the state $j$ of $p_2$. In
       which state(s), if any, is $p_1$ blocked and/or deadlocked? In which state(s), if any, is
       $p_2$ blocked and/or deadlocked?
   (c) For the state transition diagram of point (b) above, draw the resource graphs
       corresponding to the states $S_{1,0}$, $S_{1,1}$, $S_{1,2}$, $S_{1,3}$, $S_{1,4}$, $S_{1,5}$, $S_{2,3}$, $S_{3,3}$.

8. Consider the same two processes and resources as in Exercise 7, but assume now
   that both processes request $R_1$ before $R_2$. Draw the state transition diagram for the
   two processes, similar to Figure 6-4. In which state(s), if any, is $p_1$ blocked and/or
   deadlocked? In which state(s), if any, is $p_2$ blocked and/or deadlocked?

9. Consider a system of three processes and a single-resource class with four units. Each
   process needs at most two units. Show that the system is deadlock free, i.e., all states
   are safe.

10. Consider a generalization of Exercise 9 where the system consists of $n$ processes and
    a single-resource class with $m$ units. Show that the system is deadlock free if the sum
    of all maximum needs of all processes is less than $n + m$ units.

11. Prove the following by counterexamples:
   (a) A cycle is not a sufficient condition for deadlock.
   (b) A knot is not a necessary condition for deadlock in expedient state graphs.

12. Consider five processes, each running on a different machine. The wait-for graph
    contains the following edges: $p_1 \rightarrow p_2$, $p_2 \rightarrow p_3$, $p_3 \rightarrow p_4$, $p_1 \rightarrow p_5$, $p_5 \rightarrow p_4$.
    (a) Does this represent a deadlock state?
    (b) Assume process $p_5$ times out, which removes the edge $p_5 \rightarrow p_4$ from the graph.
        At the same time, $p_4$ requests a resource currently held by $p_1$.
        i. Is the new state a deadlock state? If not, could a centralized coordinator
            approach produce a phantom deadlock?
        ii. With a distributed algorithm, what probes could $p_4$ receive in response to
            making its request?
    (c) Repeat the previous problem, assuming that instead of $p_5$, process $p_1$ times out,
        which removes its two edges from the graph.
13. Consider a system that uses process rollback for recovery from deadlocks.
   (a) What are all the possible pieces of information that must be saved as part of a process checkpoint?
   (b) Can the process be restarted completely transparently from a checkpoint? If not, which actions performed or initiated by the process may be irreversible?

14. Show that resource graphs never contain directed cycles in the following two cases:
   (a) Every process must request all resources at one time.
   (b) Processes must request all resources according to a fixed ordering of all resources (Section 6.7.3).

15. Consider the following maximum claims graph:

   ![Resource Graph](image)

   (a) Show a sequence of operations leading from the given state to a deadlock state.
   (b) Show how the above deadlock could have been prevented using the banker’s algorithm (see Section 6.6.2).

16. Consider a banking system with many different accounts. Processes may transfer money between any two accounts, $A_i$ and $A_j$, by executing the following transaction:
   lock $A_i$; lock $A_j$; update $A_i$; update $A_j$; unlock $A_i$; unlock $A_j$;
   (a) Show how a deadlock can occur in such a system.
   (b) How can the ordered resource policy (see Section 6.7) be implemented to prevent deadlock if the set of possible accounts is not known a priori or changes dynamically?

17. Consider two processes, $p_1$ and $p_2$. Process $p_1$ needs resources $R_1, R_2, R_3, R_4$; process $p_2$ needs resources $R_2, R_3, R_4, R_5$. If the processes are allowed to request the resources in any order, deadlock can occur. Give five permutations of the requests that may result in deadlock, and five permutations that will never result in a deadlock.