



## Chapter 6

### 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 Linux and Windows. It is then up to the application developer to write programs that handle deadlocks.

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

To ensure that deadlocks never occur, the system can use either deadlock prevention (stopping) or a deadlock-avoidance scheme. **Deadlock prevention (not allowed)** provides a set of methods to ensure 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.

**Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system 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.

### **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 still make can be satisfied by the currently available resources plus the resources held by all  $P_j$ , with  $j < i$ . In this situation, if the resources that  $P_i$  needs are not immediately available, then  $P_i$  can wait until all  $P_j$  have finished. When they have finished,  $P_i$  can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When  $P_i$  terminates,  $P_{i+1}$  can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe*.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 7.6). An unsafe state *may* lead to a deadlock.



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.)

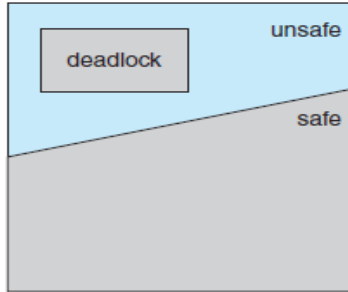


Figure 7.6 Safe, unsafe, and deadlocked state spaces.

	<u>Maximum Needs</u>	<u>Current Needs</u>	<u>maximum need to finished</u>
$P_0$	10	5	5
$P_1$	4	2	2
$P_2$	9	2	7

At time  $t_0$ , the system is in a safe state. The sequence  $\langle P_1, P_0, \text{ and } P_2 \rangle$  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 tapes 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 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.

### Claim edge.

A claim edge  $P_i \rightsquigarrow R_j$  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.

### 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 dead lock 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.

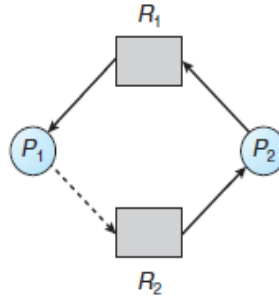


Figure 7.8 An unsafe state in a resource-allocation graph.

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.** 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.

### 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. (end of all process)
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.

### 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 (offering) to have allocated the requested resources to process  $P_i$  by modifying the state as follows:

$$\begin{aligned}
 Available &= Available - Request_i ; \\
 Allocation_i &= Allocation_i + Request_i ; \\
 Need_i &= Need_i - Request_i ;
 \end{aligned}$$

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

### An Illustrative Example



Operating System / Dr. Mahdi S. Almhanna & Dr. Rafah M. Almuttairi

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>		
	$A$	$B$	$C$	$A$	$B$	$C$	$A$	$B$	$C$
$P_0$	0	1	0	7	5	3	3	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

The content of the matrix *Need* is defined to be  $Max - Allocation$  and is as follows:

	<u>Need</u>		
	$A$	$B$	$C$
$P_0$	7	4	3
$P_1$	1	2	2
$P_2$	6	0	0
$P_3$	0	1	1
$P_4$	4	3	1

We claim that the system is currently in a safe state. Indeed, the sequence  $\langle P_1, P_3, P_4, P_2, \text{ and } P_0 \rangle$  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  $Request_1 = (1,0,2)$ . To decide whether this request can be immediately granted, we first check that  $Request_1 \leq Available$ —that is, that  $(1, 0, 2) \leq (3, 3, 2)$ , which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	<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.



### Deadlock Detection

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

### 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.

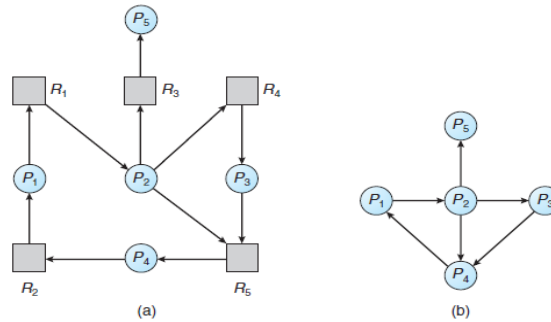


Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

More precisely, an edge from  $P_i$  to  $P_j$  in a wait-for graph implies that process  $P_i$  is waiting for process  $P_j$  to release a resource that  $P_i$  needs.

An edge  $P_i \rightarrow P_j$  exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges  $P_i \rightarrow R_q$  and  $R_q \rightarrow P_j$  for some resource  $R_q$ . In Figure 7.9, 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.

### 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





- **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<sub>i</sub>** and **Request<sub>i</sub>**. 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  $i = 0, 1, \dots, n-1$ , if **Allocation<sub>i</sub>**  $\neq 0$ , then **Finish<sub>i</sub>** = **false**. Otherwise, **Finish<sub>i</sub>** = **true**.
2. Find an index  $i$  such that both
  - a. **Finish<sub>i</sub>** == **false**
  - b. **Request<sub>i</sub>**  $\leq$  **Work**

If no such  $i$  exists, go to step 4.
3. **Work** = **Work** + **Allocation<sub>i</sub>**
  - o **Finish<sub>i</sub>** = **true**
  - c. Go to step 2.
4. If **Finish<sub>i</sub>** == **false** for some  $i, 0 \leq i < n$ , then the system is in a deadlocked state. Moreover, if **Finish<sub>i</sub>** == **false**, then process  $P_i$  is deadlocked.

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

You may wonder why we reclaim the resources of process  $P_i$  (in step 3) as soon as we determine that **Request<sub>i</sub>**  $\leq$  **Work** (in step 2b). We know that  $P_i$  is currently **not** involved in a deadlock (since **Request<sub>i</sub>**  $\leq$  **Work**). Thus, we take an optimistic attitude and assume that  $P_i$  will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked. To illustrate this algorithm, we consider a system with five processes  $P_0$  through  $P_4$  and three resource types  $A, B$ , and  $C$ . Resource type  $A$  has seven instances, resource type  $B$  has two instances, and resource type  $C$  has six instances. Suppose that, at time  $T_0$ , we have the following resource-allocation state:

	<b>Allocation</b>			<b>Request</b>			<b>Available</b>		
	<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>
$P_0$	0	1	0	0	0	0	0	0	0
$P_1$	2	0	0	2	0	2	0	0	0
$P_2$	3	0	3	0	0	0	0	0	0
$P_3$	2	1	1	1	0	0	0	0	0
$P_4$	0	0	2	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  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  results in **Finish<sub>i</sub>** == **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:

	<b>Request</b>		
	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  $P_4$ .

### 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.

### 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

### 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.
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 any practical system must address. 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.