Process Management: Deadlocks

Overview

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**.

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 non sharable 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 *{P0, P1, ..., Pn}* of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

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 = {P1, P2, ..., Pn}, the set consisting of all the active processes in the* system, and *R = {R1, R2, ..., Rm}, the set consisting of all resource types in the* system.

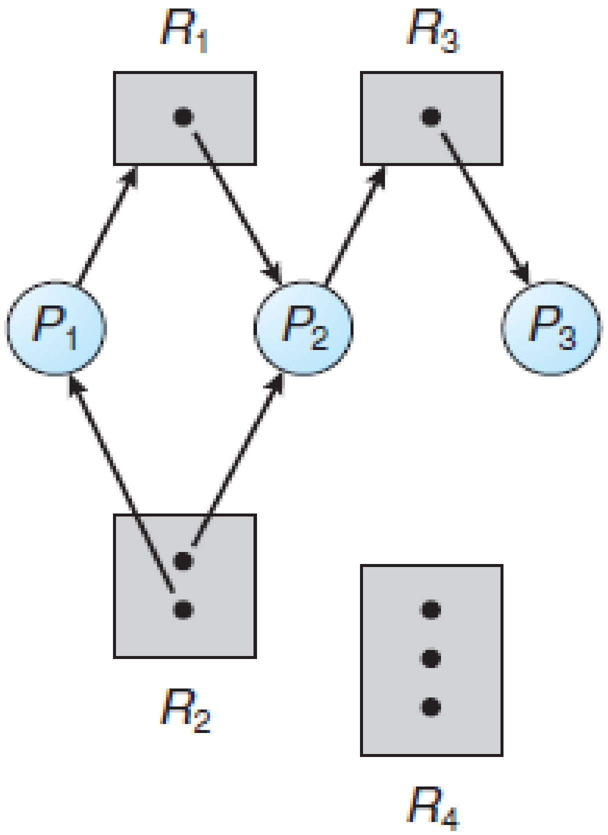


Figure 7.1 Resource-allocation graph.

- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Given the definition of a resource-allocation graph, it can be shown that,

1. If the graph contains no cycles, then no process in the system is deadlocked.
2. If the graph does contain a cycle, then a deadlock may exist.
3. If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
4. 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.
5. 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.

Examples

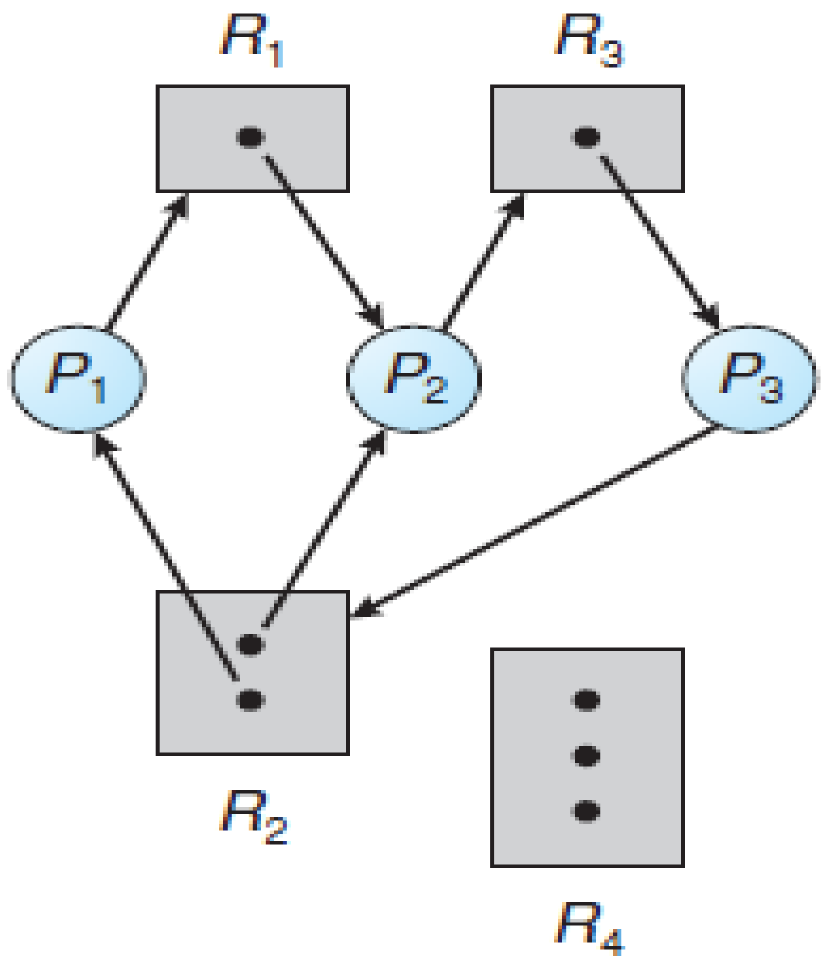


Figure 7.2 Resource-allocation graph with a deadlock.

There are two cycles in the graph:

P1 → R1 → P2 → R3 → P3 → R2 → P1
P2 → R3 → P3 → R2 → P2

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

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

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

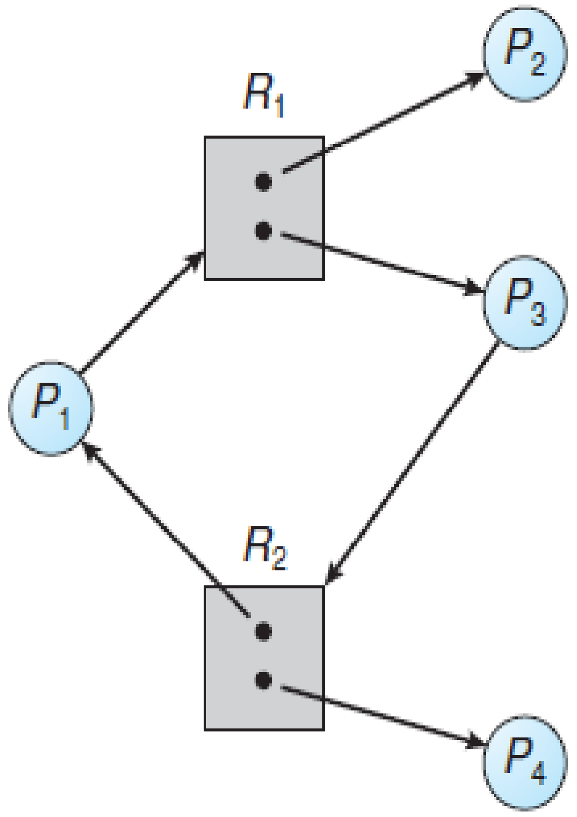


Figure 7.3 Resource-allocation graph with a cycle but no deadlock.

Methods for Handling Deadlocks

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

Deadlock Prevention

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.***

Mutual Exclusion: The mutual exclusion condition must hold. That is, at least one resource must be non sharable. 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.

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 we can use requires each process to request and be allocated all its resources before it begins execution.

Circular Wait: The another 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.

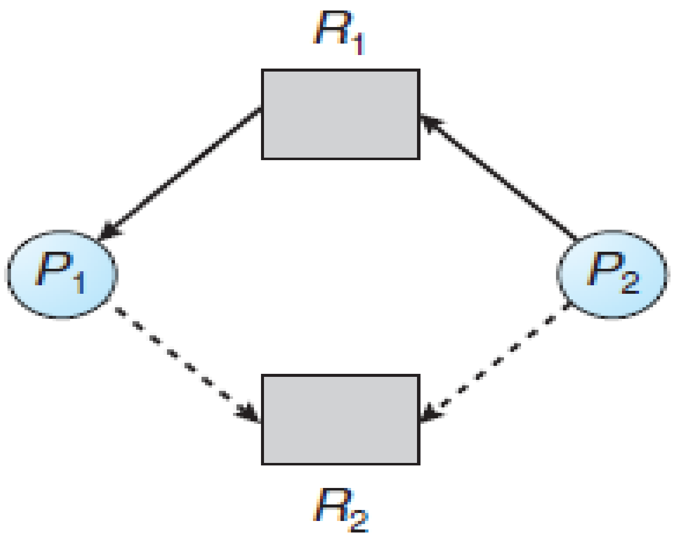
No Preemption: The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol.

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 the new ones that it is requesting.

Deadlock Avoidance

We can avoid deadlock by using a modified Resource-Allocation-Graph Algorithm.

The following is a modified graph with **request edges**, and **claim edges** (dashed lines) that indicates that process Pi may request resource Rj at some time in the future.



Now suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph.

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 Pi will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 7.7. Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph (Figure 7.8). A cycle, as mentioned, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

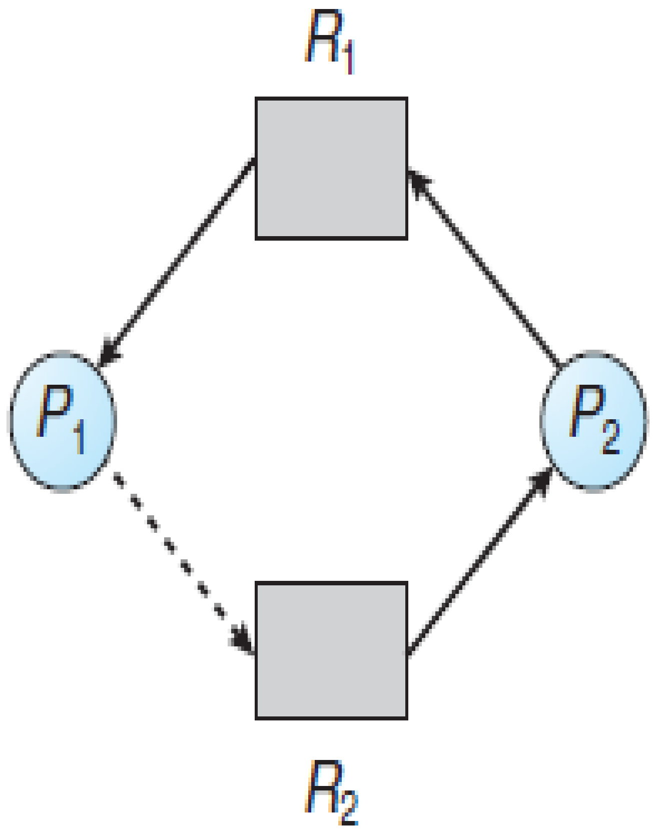


Figure 7.8 An unsafe state in a resource-allocation graph.

Banker’s Algorithm

The banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “safe-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

Banker’s Algorithm (Count..)

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.

Banker’s Algorithm (Count..)

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:

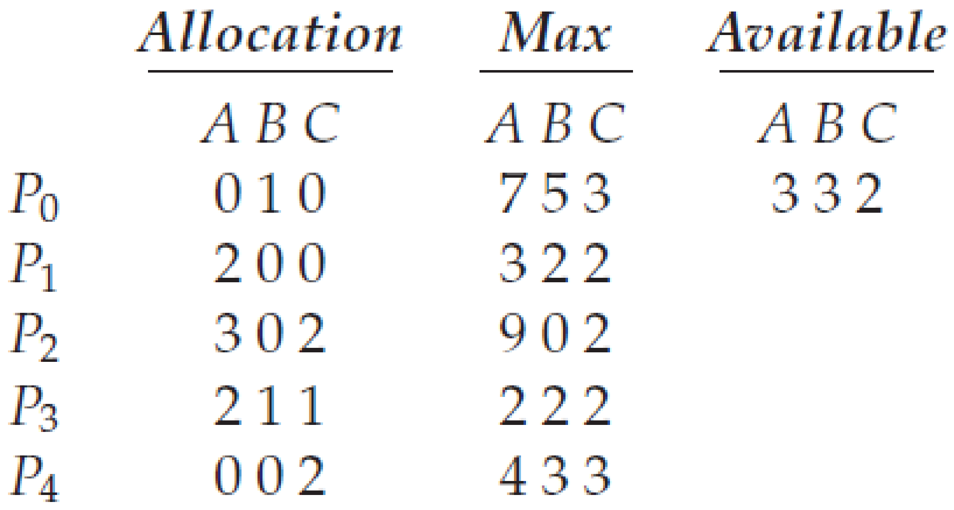
Available. Avector of length*m*indicates the number of available resources of each type. If ***Available***[*j*] equals *k*, then *k* instances of resource type *Rj* are available.

- **Max**. An *n* × *m* matrix defines the maximum demand of each process. If ***Max***[*i*][*j*] equals *k*, then process *Pi* may request at most *k* instances of resource type *Rj* .
- **Allocation**. An *n* × *m* matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***[*i*][*j*] equals *k*, then process *Pi* is currently allocated *k* instances of resource type *Rj* .
- **Need**. An *n* × *m* matrix indicates the remaining resource need of each process. If ***Need***[*i*][*j*] equals *k*, then process *Pi* may need *k* more instances of resource type *Rj* to complete its task. Note that

$$\textit{\textbf{Need}}[i][j] = \textit{\textbf{Max}}[i][j] - \textit{\textbf{Allocation}}[i][j]$$

Banker’s Algorithm illustration

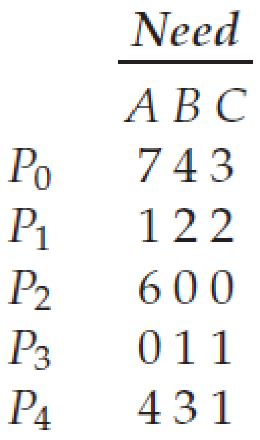
To illustrate the use of the banker’s algorithm, consider a system with five processes *P*0 through *P*4 and three resource types *A*, *B*, and *C*. Resource type *A* has ten instances, resource type *B* has five instances, and resource type *C* has seven instances. Suppose that, at time *T*0, the following snapshot of the system has been taken:



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

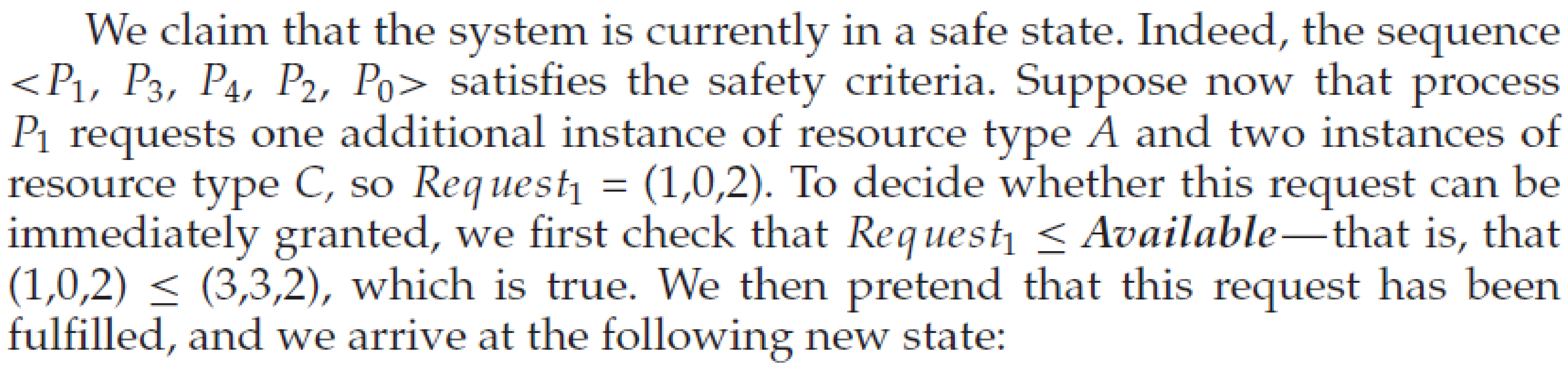
Banker's Algorithm illustration (Cout..)

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



Need

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



We claim that the system is currently in a safe state. Indeed, the sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies the safety criteria. Suppose now that process P_1 requests one additional instance of resource type *A* and two instances of resource type *C*, so $Request_1 = (1,0,2)$. To decide whether this request can be immediately granted, we first check that $Request_1 \leq$ ***Available***—that is, that $(1,0,2) \leq (3,3,2)$, which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

Banker’s Algorithm illustration (Cout..)

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

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

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

We leave it as a programming exercise for students to implement the banker’s algorithm.