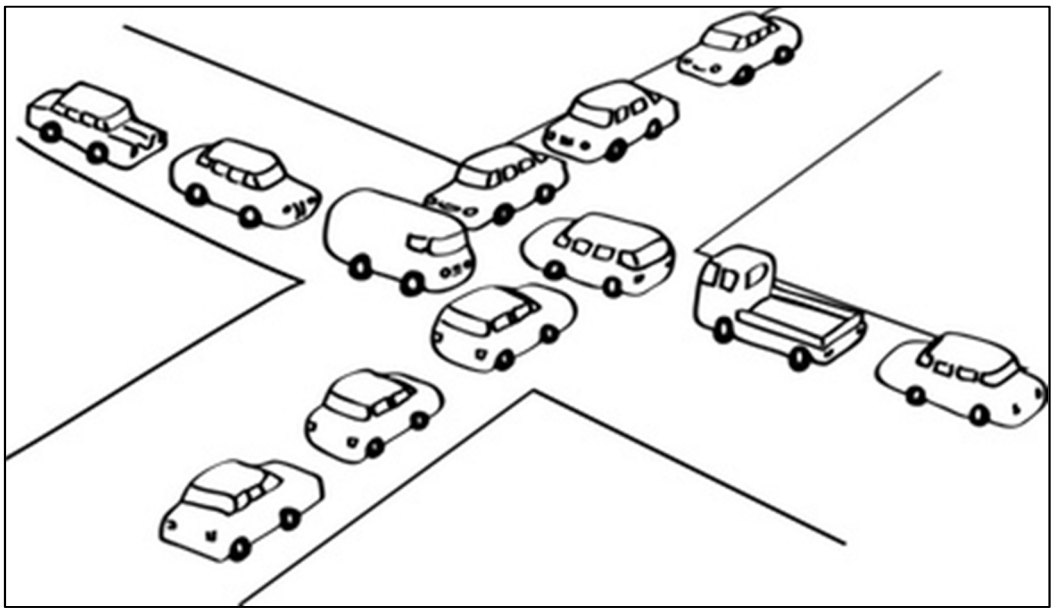
Chapter 9
Deadlocks

This chapter will discuss the following concepts:

9.1 The Deadlock Problem
9.2 System Model
9.3 Deadlock Characterization
9.4 Methods for Handling Deadlocks
9.5 Deadlock Prevention
9.6 Deadlock Avoidance
9.7 Deadlock Detection
9.8 Recovery from Deadlock



9.1 The Deadlock Problem

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

Example1:

A System has 2 disk drives. Process P1and P2 each holds one disk drive and each needs another one.

Example2: in figure 9.1

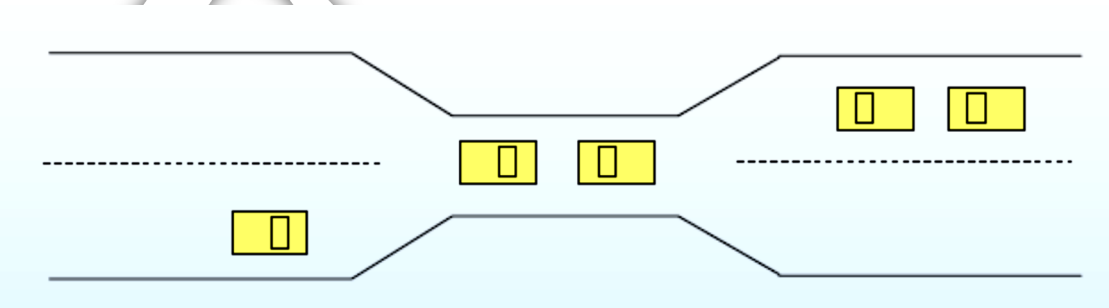


Figure 9.1 deadlock example

1. Traffic only in one direction.
2. Each section of a bridge can be viewed as a resource.
3. If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
4. Several cars may have to be backed up if a deadlock occurs.

9.2 System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned

into several types, each consisting of some number of identical instances. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request: If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. Use: The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. Release: The process releases the resource.

9.3 Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. We look more closely at features that characterize deadlocks.

9.3.1 Necessary Conditions

A deadlock situation can arise if the following four conditions hold **simultaneously** in a system, we emphasize that **all four conditions** must hold for a deadlock to occur:

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.

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 by the process holding it, after that process has completed its task.

4. Circular wait: A set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, …., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

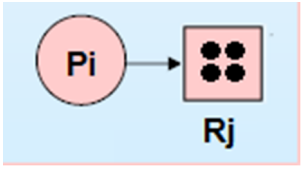
9.3.2 Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system **resource-allocation graph**.

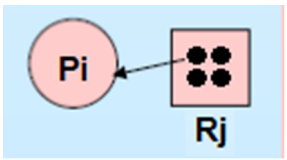
- This graph consists of a set of vertices V and a set of edges E.
- The set of vertices V is partitioned into two different types of nodes: $P = \{P_1, P_2, ……, P_n\}$ the set consisting of all the active processes in the system, and $R = \{R_1, R_2, …, R_m\}$, the set consisting of all resource types in the system.
- We represent each process Pi, as a circle

- We represent each resource type Rj as a rectangle. Since resource type Rj may have more than one instance, we represent each such instance as a dot within the rectangle.

- A directed edge called a **request edge** from process Pi to resource type Rj is denoted by Pi → Rj; it signifies that process Pi, has **requested** an instance of resource type Rj, and is currently waiting for that resource.



- A directed edge called an **assignment edge** from resource type Rj to process Pi is denoted by Rj → Pi; it signifies that an instance of resource type Rj has been **allocated** to process Pi; An assignment edge must also designate one of the dots in the rectangle (instance).



- When process Pi, requests an instance of resource type Rj, a **request edge** is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an **assignment edge**.
- When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.
- Given the definition of a resource-allocation graph, it can be shown that, if the graph contains **no cycles**, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock **may exist**.
- If each resource type has exactly **one instance**, then a cycle implies that a deadlock has occurred. Each process involved in the cycle is deadlocked.
- If each resource type has **several instances**, then a cycle does not necessarily imply that a deadlock has occurred.

Example1:

The resource-allocation graph shown in Figure 9.2 depicts the following situation.

- The sets P, R, and E:
  - P={P1,P2,P3}
  - R= {R1, R2, R3, R4}
  - E = {p1 →R1, P2 → R3, R1 → p2, R2 → P2, R2 → p1, R3 →P3 }
- Resource instances:
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instances of resource type R4

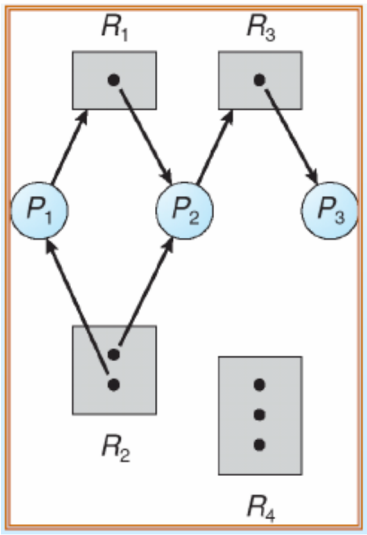


Figure 9.2 Resource-allocation graph.

- Process states:
  - Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
  - Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
  - Process P3 is holding an instance of R3.

Example2:

To illustrate this concept, we return to the resource-allocation graph depicted in Figure 9.2. Suppose that process **P3** requests an instance of resource type **R2**. **Since no resource instance is currently available**, a request edge P3 → R2 is added to the graph (Figure 9.3). At this point, two minimal cycles exist in the system:

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

Therefore 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.

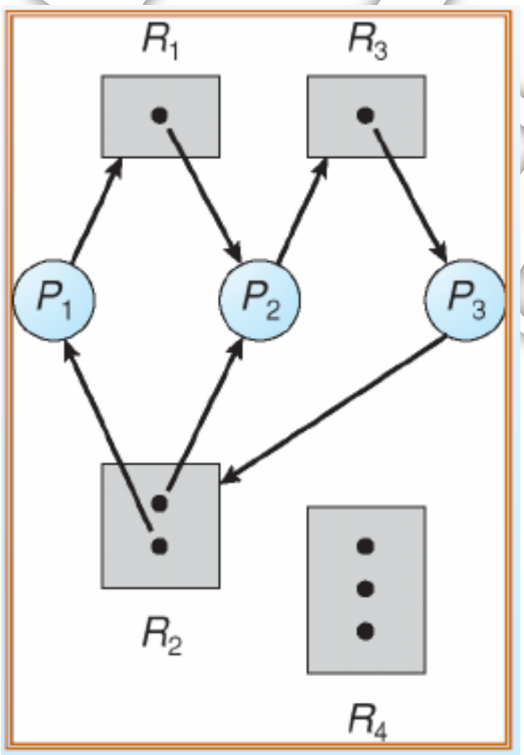


Figure 9.3 Resource-allocation graph with a deadlock.

Example3:

Now consider the resource-allocation graph in Figure 9.4. In this example, we also have a cycle

P1 →R1→ P3 →R2→ P1

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle,

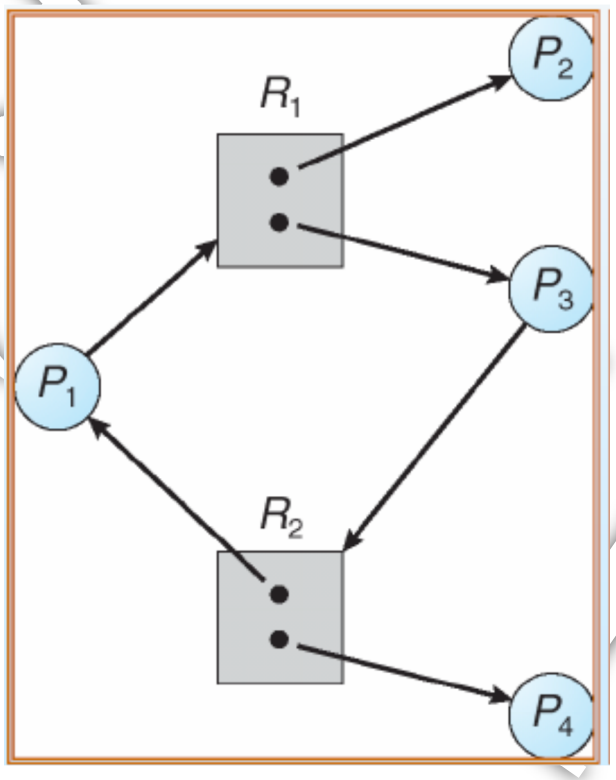


Figure 9.4 Resource-allocation graph with a cycle but no deadlock.

In summary if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

9.4 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 deadlock state.
- We can allow the system to enter a deadlock state, **detect** it, and **recover**.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

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

9.5 Deadlock Prevention

As we noted in Section 9.3.1, for a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

9.5.1 Mutual Exclusion

The mutual-exclusion condition must hold for **non-sharable resources**. For example, a printer cannot be simultaneously shared by several processes. **Sharable resources**, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. In general, however, we cannot prevent deadlocks by denying the mutual-

exclusion condition, because some resources are intrinsically non-sharable.

9.5.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it **does not hold** any other resources.

9.5.3 No Preemption

The third necessary condition for deadlocks is that there is 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, then all resources currently being held are implicitly released. The released 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.

9.5.4 Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

For example, if the set of resource includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F (tape drive) = 1

F (disk drive) = 5

F (printer) = 12

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

9.6 Deadlock Avoidance

With the knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. The resource-allocation state is defined by the number of **available** and **allocated** resources and the **maximum** demands of the processes.

9.6.1 Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**.

- A sequence of processes <P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi, can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i. If the resources that Pi needs are not immediately available, then Pi, can wait until all Pj have finished. When pj have finished, Pi; can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.
- When Pi terminates, Pi+l 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

Example:

Consider a system with 12 magnetic tape drives and three processes: P0, P1, and P2. Process P0 requires 10 tape drives, process P1 may need as many as 4 tape drives, and process P2 may need up to 9 tape drives. Suppose that, at time to, process P0 is holding 5 tape drives, process P1 is holding 2 tape drives, and process P2 is holding 2 tape drives. (Thus, there are **3 free** tape drives.)

Process	Maximum need	Allocated
P_0	10	5
P_1	4	2
P_2	9	2

At time t0, the system is in a safe state. The sequence < P1, P0, p2> satisfies the safety condition.

- Process P1 can immediately be allocated all its tape drives and then return them (the system will then have (3+2) = 5 available tape drives);
- Then process P0 can get all its tape drives and return them (the system will then have (5+5) =10 available tape drives);
- And finally process P2 can get all its tape drives and return them (the system will then have all 12 tape drives available).

A system can go from a safe state to an unsafe state.

Suppose that, at time T1, process P2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P1, can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives.

9.6.2 Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with **multiple instances** of each resource type. The deadlock avoidance algorithm that we describe next is applicable to such a system.

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.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let **n** be the number of processes in the system and **m** be the number of resource types. We need the following data structures:

- **Available:** A vector of length m indicates the number of available resources of each type. If Available [j] equals k, there are k instances of resource type Rj available.
- **Max:** An n x 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 R j.
- **Allocation:** An n x 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 R j.

- **Need:** An n x 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 R j to complete its task.

Need[i][j] = Max[i][j] - Allocation[i][j].

9.6.2.1 Safety Algorithm

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

1. Let *Work* and *Finish* be vectors of length *m* and *n,* respectively. Initialize
   Work = available and Finish[i] = false for i=0, 1, ..., *n – 1.*
2. Find an i such that both
   a. *Finish[i] ==false*
   b. *Need i <= Work*
   If no such i exists, go to step 4.
3. Work = Work + Allocation i
   Finish[i] = true
   Go to step 2.
4. If Finish[i] = true for all i, then the system is in a safe state.

9.6.2.2 Resource-Request Algorithm

We now describe the algorithm which determines if requests can be safely granted. Let Request i be the request vector for process Pi. If Request i [j] == k, then process Pi, wants k instances of resource type R j.

When a request for resources is made by process Pi the following actions are taken:

1. If *Request i < Need i,* go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If *Request <= Available,* go to step 3. Otherwise, *Pi* must wait, since the resources are not available.

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

Available = Available – Request i
Allocation i= Allocation i + Request i
Need i = Need i- Request i

9.6.2.3 An Illustrative Example

Consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at time T0, the following snapshot of the system has been taken:

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

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

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

We claim that the system is currently in a safe state. Indeed, the sequence <P1, P3, P4, P0, P2> satisfies the safety criteria.

Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request = (1,0,2). To decide whether this request can be immediately granted, we first check that Request < Available—that is, that (1,0,2) < (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

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

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <P1, P3, P4, P0, P2> satisfies the safety requirement. Hence, we can immediately grant the request of process P1.

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are **not available**. Furthermore, a request for (0,2,0) by Po cannot be granted, even though the resources are available, since the resulting state is unsafe.

9.7 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 must examine the state of the system to

determine whether a deadlock has occurred then to recover from the deadlock.

9.7.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for graph**. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

For example, in Figure 9.5, 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.

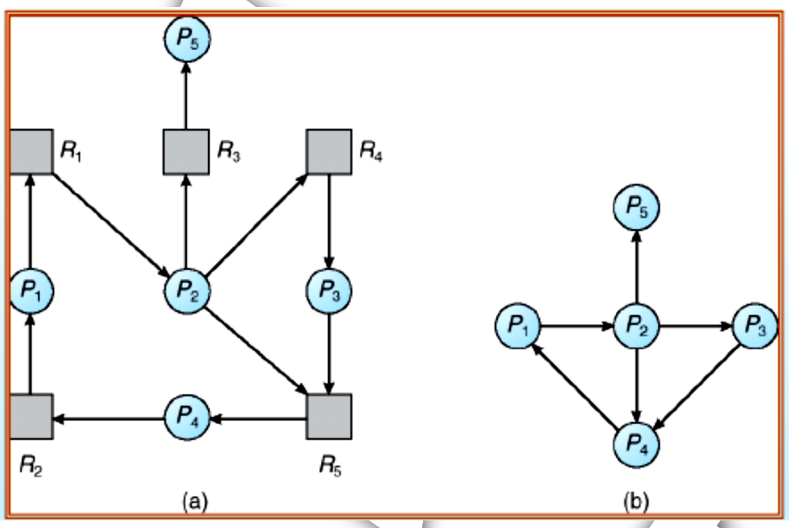


Figure 9.5 (a) Resource-allocation graph, (b) Corresponding wait-for graph.

9.7.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm (Section 9.6.2):

9.8 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.

End of chapter 9