DEADLOCKS

INTRODUCTION

- Deadlock: ***"A lock having no keys"***
- *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.*
- 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.
- A process may utilize a resource in only the following sequence:
  1. **Request**. The process requests the resource. If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
  2. **Use**. The process can operate on the resource
  3. **Release**. The process releases the resource.
- A **system table** records whether each resource is free or allocated; for each resource that is allocated, the table also records the process to which it is allocated. If a process

requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

- **Example for deadlock**: consider a system with one printer and one DVD drive. Suppose that process *Pi* is holding the DVD and process *Pj* is holding the printer. If *Pi* requests the printer and *Pj* requests the DVD drive, a deadlock occurs.

NECESSARY CONDITIONS FOR DEADLOCK

- For occurring deadlocks, 4 necessary conditions should be satisfied:

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 , Pl, ... , Pn} of waiting processes must exist such that *Po* 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 *Po.*

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.
- A directed 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 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.
- **A directed edge Pi→Rj is called a request edge; a directed edge Rj→Pi is called an assignment edge.**
- We represent each process *Pi* as a **circle** and each resource type *Rj* as a **rectangle**. Since resource type *Ri* may have more than one instance, we represent each such instance as a **dot** within the rectangle.

- A request edge points to only the rectangle *Rj*, whereas an assignment edge must also designate one of the dots in the rectangle.
- 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.
- Consider the following resource-allocation graph

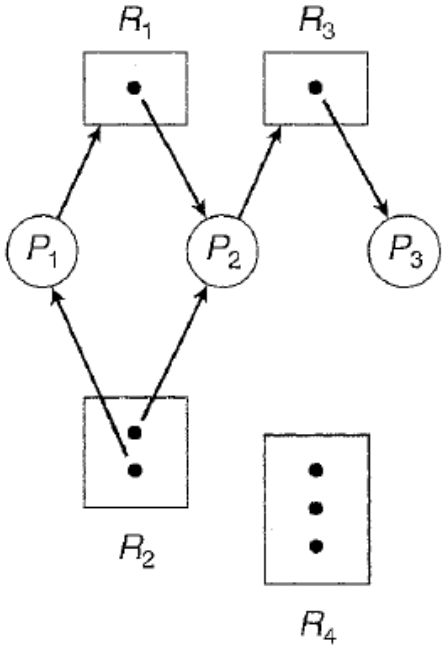


Figure 7.2 Resource-allocation graph.

- 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.

- **If the graph contains no cycles, then no process in the system is deadlocked.**
- If the graph contains a cycle, then a deadlock may exist.
- **If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.**
- **In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.**
- **If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.**
- Eg1:

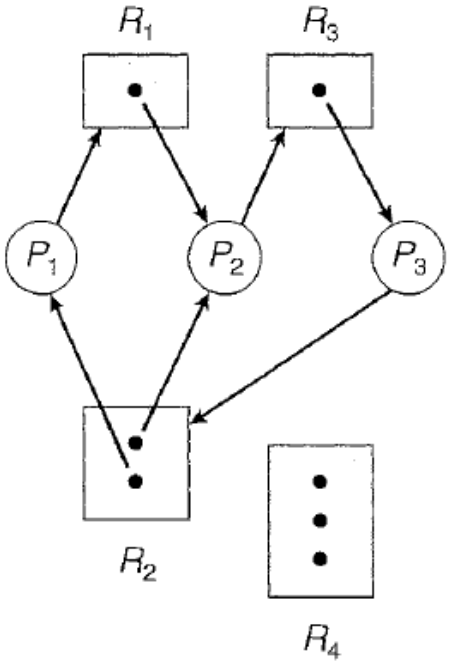


Figure 7.3 Resource-allocation graph with a deadlock.

- There are 2 cycles:
  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.
- Another Eg2:

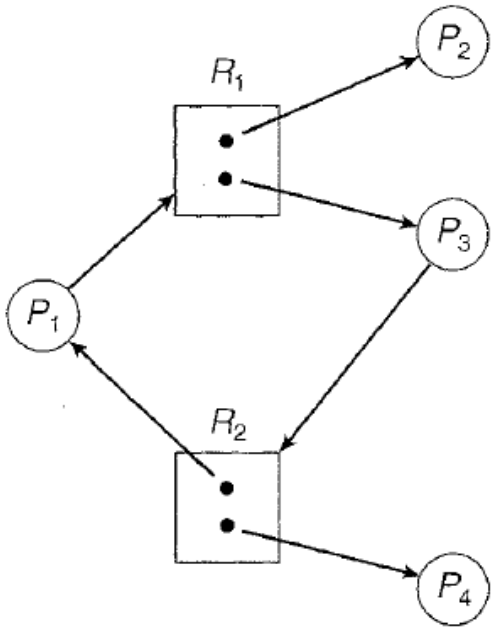


Figure 7.4 Resource-allocation graph with a cycle but no deadlock.

- Here also we 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.
- 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.

METHODS FOR HANDLING DEADLOCKS

1. **Prevent or avoid deadlocks** ensuring that the system will *never* enter a deadlocked state.
2. Allow the system to enter a deadlocked state, **detect it, and recover it**.
3. **Ignore the deadlock problem** and pretend that deadlocks never occur in the system.

- The third solution is the one used by most OS, including UNIX and Windows
- It is then up to the application developer to write programs that handle deadlocks.
- In this case, the undetected deadlock will affect system's performance
- Eventually, the system will stop functioning and will need to be restarted manually.
- In many systems, **deadlocks occur infrequently (say, once per year)**. So, this method is cheaper than the prevention, avoidance, or detection and recovery methods.

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.

1. Mutual Exclusion

- The mutual-exclusion condition must hold for non-sharable resources.
- **Sharable resources** 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.
- However, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are basically non-sharable.

2. Hold and Wait

- To ensure that the hold-and-wait condition never occurs in the system, guarantee that, whenever a process waits for a resource, it does not hold any other resources.
- **One method** that can be used requires each process to **request and be allocated all its resources before it begins execution.**
- An **alternative method** allows a process to **request some resources only when it has no resources.** ie **Before it can**

request any additional resources it must release all the resources that it is currently allocated.

- Example: Consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.
- If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.
- The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file.
- The process must then again request the disk file and the printer. After printing, it releases these two resources and terminates.
- **Disadvantages**: In **first method, resource utilization may be low**, since resources may be allocated but unused for a long period.
- In **second method**, we can release the DVD drive and disk file, and then **again request** the disk file and printer only if we can be **sure that our data will remain on the disk file**.ie no other process changes the data in disk file.

3. No Preemption

- To ensure that this condition does not hold, **we can use preemption**.
- 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.
- If a process requests some resources, first check whether they are available.
- If available allocate them.
- If they are not available, check whether they are allocated to some other process that is waiting for additional resources.
- If so, **preempt the desired resources from the waiting process and allocate them to the requesting process.**
- If the resources are neither available nor held by a waiting process, the requesting process must wait. **While it is waiting, some of its resources may be preempted, only if another process requests them.**
- The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

4. Circular Wait

- Ensure that this condition never holds

- Impose **a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.**
- Let *R* = {R1, R2, ... ,Rm} be the set of resource types.
- Assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.
- Define a one-to-one function F: *R* →*N*, where *N* is the set of natural numbers.
- Example,

  F (tape drive) = 1

  F (disk drive) = 5

  F (printer) = 12
- **Each process can request resources only in an increasing order of enumeration.**
- A process can initially request any number of instances of a resource type *Ri.*
- After that, the process can request instances of resource type *Rj* if and only if F(Rj) >F(Ri).
- A process requesting an instance of resource type *Rj* must have released any resources *Ri* such that F(Ri)>=F(Rj).
- Let the set of processes involved in the circular wait be {*P0* , P1, ... , Pn,}where *Pi* is waiting for a resource *Ri*, which is held by process *Pi+l*·
- Since process *Pi+1* is holding resource *Ri* while requesting resource *Ri+1*, we must have F(Ri) < F(Ri+1) for all *i.*

- But this condition means that F(R0) < F(R1) < ... < F(Rn) <*F* (R0).
- By transitivity, F(R0) < F(R0), which is impossible. Therefore, there can be no circular wait.
- **Eg: Consider Process P1 holds resource number 1 and waits for resource number 2. Process P2 holds resource number 2 and waits for resource number 3. Process P3 holds resource number 3 but cannot wait for resource number 1, since 1<3. So there should not be a circular waiting situation, if we follow the increasing order of resources.**

DEADLOCK AVOIDANCE

- It requires **additional information about how resources are going to be requested in future**.
- When a process request for a resource, the system should be able to **decide whether it can be granted or not.**
- In this decision making, the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
- Each process declares the *maximum number* of resources that it may need.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.

- The following algorithms are used for deadlock avoidance

1. Safe State Algorithm

- A state is *safe* if the system can allocate resources to each process in some order and still avoid a deadlock.
- **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 if, for each *Pi*, the resource requests by *Pi* 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 they 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.**
- **A deadlocked state is an unsafe state.**
- **Not all unsafe states are deadlocks; unsafe state *may* lead to a deadlock.**

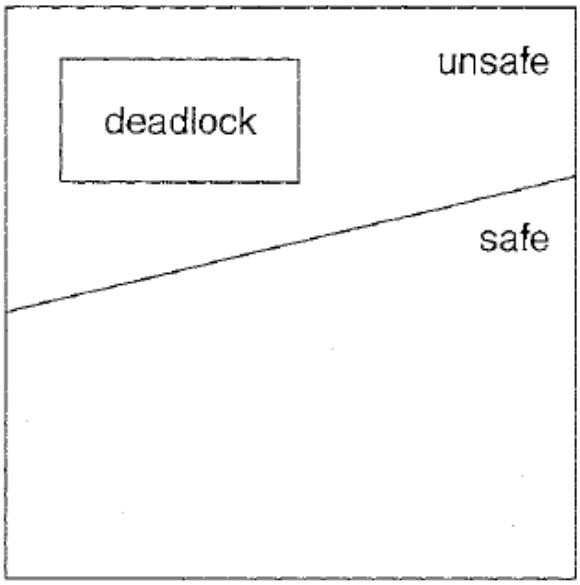


Figure 7.5 Safe, unsafe, and deadlocked state spaces.

- Example: Consider a system with twelve magnetic tape drives and three processes: Po, P1, and P2. Process Po requires ten tape drives, process P1 may need four tape drives, and process P2 may need up to nine tape drives.
- Suppose currently process Po is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives.

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- The system is in a safe state. The sequence <P1, P0, P2> satisfies the safety condition.

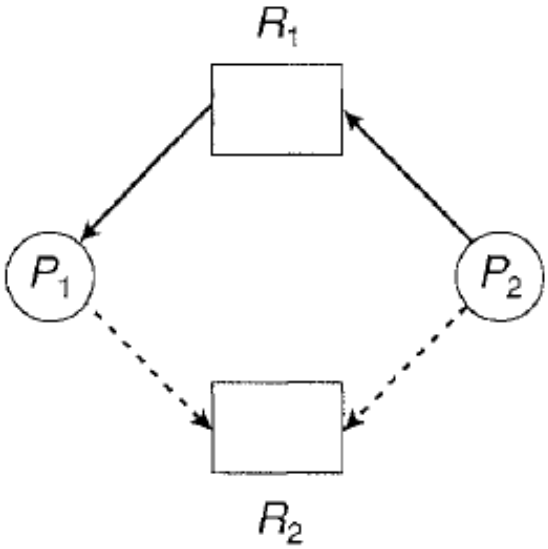
- Process P1 can immediately be allocated its free tape drives (2 more) and then return them
- Then process Po can get all its tape drives and return them (1 free + 4 from P1 = total 5)
- Finally process P2 can get all its tape drives and return them (all are free now)
- **A system can go from a safe state to an unsafe state suddenly.**
- Suppose process P2 requests and is allocated one more tape drive.
- The system is no longer in a safe state.
- After allocating one more tape drive to P2, there are 2 more free.
- Process P1 can be allocated with these 2 and complete it.
- When it returns them, the system will have only four available tape drives.
- Po needs five more tape drives. It will have to wait, because they are unavailable.
- Similarly, process P2 may request six additional tape drives and have to wait, resulting in a deadlock.
- Our mistake was in granting the request from process P2 for one more tape drive.
- If we had made P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

- **The request is granted only if the allocation leaves the system in a safe state.**
- **Drawback**: If a process requests a resource that is currently available, it may still have to wait. Thus, **resource utilization may be lower**

2. Resource-Allocation-Graph Algorithm

- **Applicable to the systems with only one instance of each resource.**
- In a resource allocation graph, in addition to the request and assignment edges we introduce a **new type of edge, called a claim edge.**
- **A claim edge *Pi* →*Rj* indicates that process *Pi* may request resource *Rj* at some time in the future**.
- This edge resembles a request edge in direction but is represented in the graph **by a dashed line.**
- When process *Pi* requests resource *Rj*, the claim edge *Pi*→*Rj* is converted to a request edge.
- Before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.
- 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.

- We check for safety by using a **cycle-detection algorithm. It requires an order of n^2 operations, where n is the number of processes in the system.**
- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process *Pi* will have to wait for its requests to be satisfied.
- Consider the resource-allocation graph below:



- Suppose *P2* requests R2. Although R2 is currently free, we cannot allocate it to *P2,* since this action will create a cycle in the graph
- A cycle indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.
- **Drawback**: **Applicable to the systems with only one instance of each resource.**

3. Banker's Algorithm

[Will be discussed later]

DEADLOCK DETECTION

- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.
- An algorithm examines the state of the system to determine whether a deadlock has occurred or not

1. Single Instance of Each Resource

- 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.**
- As before, a **deadlock exists in the system if and only if the wait-for graph contains a cycle**
- 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.

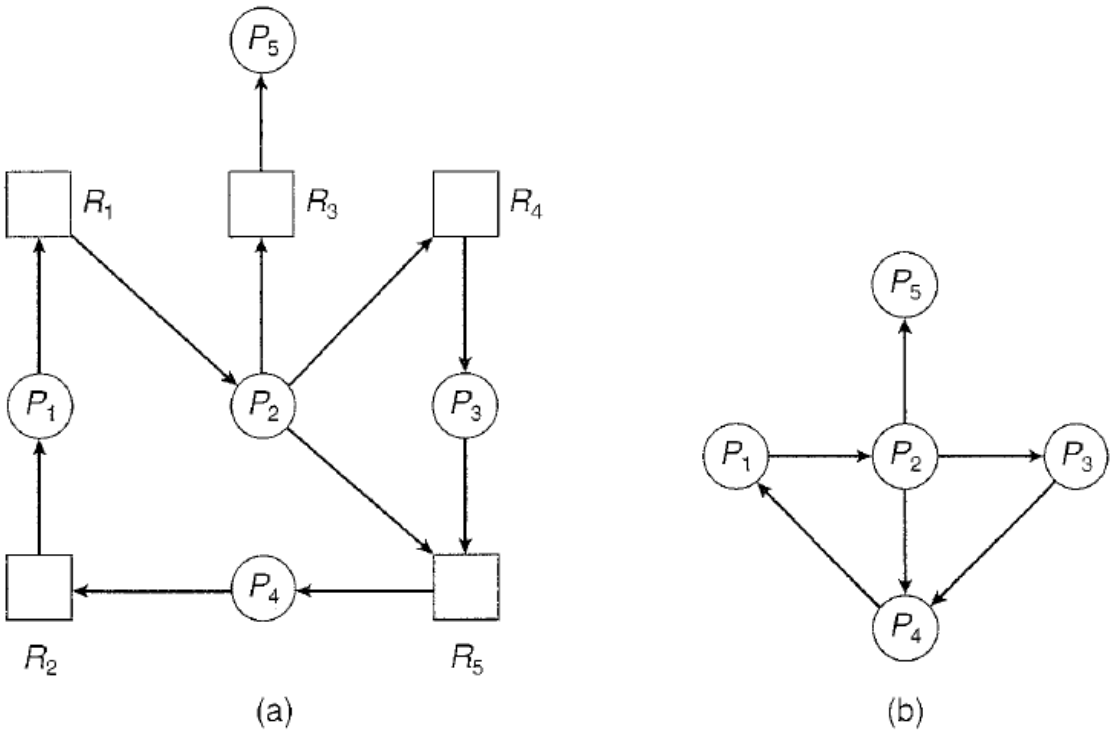


Figure 7.8 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

- **Drawback:** The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

2. Several Instances of a Resource

- **Deadlock detection algorithm** is used
- **Similar to Bankers algorithm**
- In Bankers algorithm (deadlock avoidance), future information (***Max*** & ***Need***) is also considered. But during deadlock detection, future information is not needed. Only the present request is considered.
- All data structures in detection algorithms are same as that of Bankers algorithm except in the case of ***Max*** & ***Need***.
- Here we use a matrix ***Request*** instead of ***Need***.

- **Deadlock detection Algorithm**:
  [Write Bankers algorithm by renaming Need as Request]
- Example problem
  [Will be discussed later]

RECOVERY FROM DEADLOCK

- When a detection algorithm determines that a deadlock exists either it may be handled **manually** by the user or recovered **automatically** by the OS
- 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 Abortion

- 2 ways are there for process abortion

1. **Abort all deadlocked processes.**
   - This method clearly will break the deadlock cycle, but at great expense.
   - All the work done by these processes become invalid and to be recomputed again
2. **Abort one process at a time until the deadlock cycle is eliminated**.
   - After aborting each process, deadlock detection algorithm is run.

- We must determine which deadlocked process should be aborted.
- This determination is a policy decision, similar to CPU-scheduling decisions.
- **We should abort those processes whose termination will incur the minimum cost.**
- Many factors may affect which process is chosen:
  - What the priority of the process is?
  - How long the process has computed and how much longer the process will compute before completing its designated task
  - How many and what types of resources the process has used
  - How many more resources the process needs in order to complete
  - How many processes will need to be terminated
  - Whether the process is interactive or batch

- **Drawback of process abortion**: If the process was in the middle of updating a file, aborting it will leave that file in an incorrect state.

Resource Preemption

- We can successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- Three issues need to be addressed:

- **Selecting a victim.** Which resources and which processes are to be preempted? We must determine the order of preemption to **minimize cost**. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.
- **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**.
- **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?