UNIT-III

Dead locks

Definition:

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 Dead Lock.

System model

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

A process may utilize a resource in only in the following sequence.

1. **Request**. The process requests the resource. 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.

A process must request a resource before using it and must release the resource after using it.

Dead lock characterization

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 $\{P_0, P_1, ..., P_n\}$ 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.

Resource Allocation Graph.

Dead locks can be described more precisely in terms of a directed graph called a System resource allocation graph.

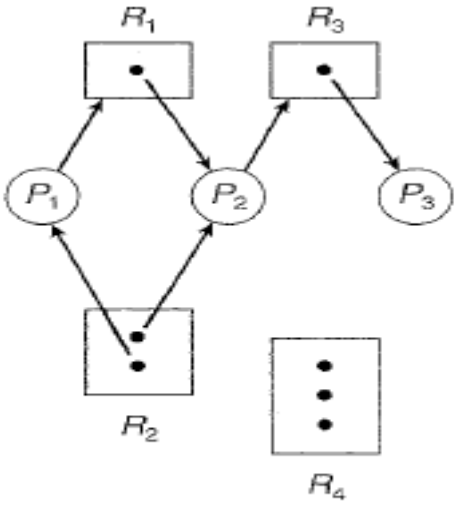

Figure 7.2 Resource-allocation graph.

A directed edge from process P_i to resource type R_j is deonted P_i -> R_j.

It signifies that process P_i has requested an instance of resource type R_j and it is called “Request Edge”.

The directed edge from resource type R_j to process P_i is denoted by R_j -> P_i.

It signifies that an instance of resource type Rj has been allocated to process Pi and it is called as "Assignment Edge".

We represent each process as a circle. And each resource type as rectangle.

Each resource type may have more than one instance. We represent each instance as a dot(.) with in a rectangle.

Note that a request edge points to only the rectangle Rj where as assignment edge must also designate one of the dots in rectangle.

- The sets P, R, and E:
  - $P = \{P_1, P_2, P_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Resource instances:
  - One instance of resource type R_1
  - Two instances of resource type R_2
  - One instance of resource type R_3
  - Three instances of resource type R_4
- Process states:
  - Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
  - Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
  - Process P_3 is holding an instance of R_3.

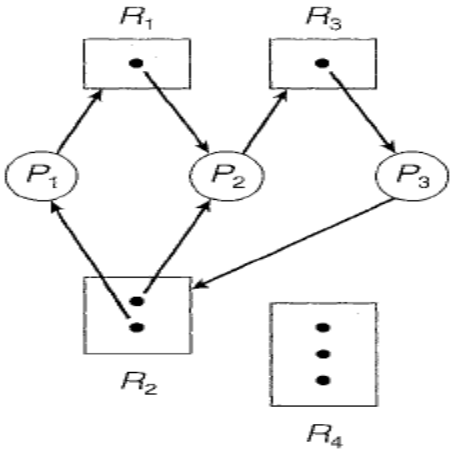

Figure 7.3 Resource-allocation graph with a deadlock.

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

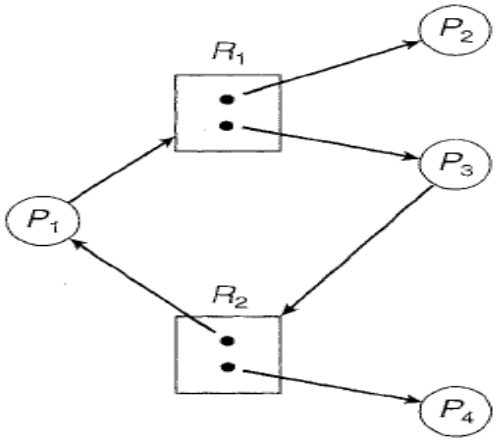

Figure 7.4 Resource-allocation graph with a cycle but no deadlock.

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

Now consider this graph, we have a cycle . However there is no dead lock, observe that p4 may release its instance of resource type R2, that resource can then be allocated to P3 breaking the cycle.

If resource allocation graph does not have a cycle, then the system is not dead locked state. If there is a cycle then the system may or may not be in a dead locked state.

Methods for Handling Dead locks

1. DEAD LOCK PREVENTION OR DEAD LOCK AVOIDANCE
2. DEAD LOCK DETECTION OR RECOVERY FROM DEAD LOCK
3. IGNORE THE DEAD LOCKS

We can deal with the dead lock problem in one of 3 ways.

1. We can use a protocol to prevent or Avoid dead locks ensuring that the system will never enter a dead locked state.
2. We can allow the system to enter a dead locked state, Detect it and Recover
3. We can ignore the problem all together and pretend that Dead locks occur in the system.

   Ex: Unix and Windows

Dead Lock Prevention

For a dead lock to occur each of the four necessary conditions must hold. By ensuring that atleast one of these conditions cannot hold. We can prevent the occurrence of a Dead lock.

i) **Mutual Exclusion**: This condition must hold non-sharable resources. Example a printer cannot be simultaneously shared by several processes.

Sharable resources do not require mutual exclusive access and thus can not be involved in a dead lock. Example Read only files are a good example of a sharable resources.

ii) **Hold and Wait**: To ensure that, this condition never occurs in systems we must guarantee that whenever a process requests a resource, it does not hold any other resources.

There are 2 protocols

One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. Disadvantage low resource utilization.

An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them before it can request any additional resources, it must release all the resources that it is currently allocated.

Ex. DVD, HARD DISK,PRINTER

Consider a process that copies data from DVD to a file on disk,sorts the file and then prints the results through a printer. Disadvantage we require all resources, it leads to starvation.

iii) **No Preemption**: This condition is 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 request another resource that can not be immediately allocated to it. Then all resources the process is currently holding are preempted.

If a process requests some resources we first check whether they are available. If they are then allocate them. If they are not, then we check whether they are allocated to some other process. Ie,. Waiting for

additional resources, if so we preempt the desired resources from waiting process and allocate them to the requesting process.

iv) **Circular wait**: that this condition never holds is to impose a total ordering of all resource types and to require that each process request resources in an increasing order of enumeration. To illustrate let R={R1,R2,..Rm} be the set of resource types, we assign to each resource type a unique integer number , which allows us to compare 2 resources and to determine whether one precedes another in our ordering . we define a one-one function

F: R->N where N is natural number

F(TAPE DRIVE)=1

F(DISK DRIVE)=5

F(PRINTER)=12

A process can initially request any number of instances of a resource type say R_i. After that The process can request instances of resources types R_j, iff

$F(R_j) > F(R_i)$

For example 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.

Dead lock Avoidance

The various algorithms that we use in this approach differ in the amount and type of information required. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.

Given this a priori information it is possible to construct an algorithm that ensures that the system will never enter the dead lock state.

The dead lock avoidance algorithm dynamically examines the resource allocation state. This resource allocation state is defined by the number of available and allocated resources and the maximum demands of processes.

i) Safe State

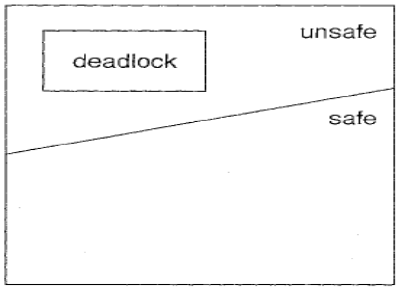

Figure 7.5 Safe, unsafe, and deadlocked state spaces.

A state is safe if the system can allocate resources to each process in some order and still avoid a dead lock. A system is in a safe state only if there exists a safe sequence. If no such sequence exists then the system state is said to be unsafe. An unsafe state may lead to a dead lock.

For example there are total 12 magnetic tape drives and 3 processes(P_0,P_1,P_2)

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

Suppose at time zero (to) process P0 is holding 5 tape drives, P1 holds 2 tape drives, P2 holds 2 tape drives.

P1 can immediately be allocated all its tape drives and then return them(5 Available).

Then process P0 can get all its tape drives and return them(10 available)

Finally P2 can get all its tape drives and return them (the system have 12 tape drives available)

At time t0, the system is in a safe state. The sequence <P1,P0,P2> satisfies the safety condition.

Suppose that at time t1, process P2 request and is allocated one more tape drive. Ie., 2 tape drives given to P1 and 1 tape drive given to P2 then P1 releases only 4 tape drives. That is not sufficient to any of the process nor P0(5 tape drives required) and P2(required 6 tape drives), so we get unsafe condition Ie., deadlock.

Whenever a process requests a resource that is currently available , the system must decide whether the resource can allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

ii) Resource Allocation Graph Algorithm

If we have a resource allocation system with only one instance of each resource type, we can use a variant of the resource allocation graph.

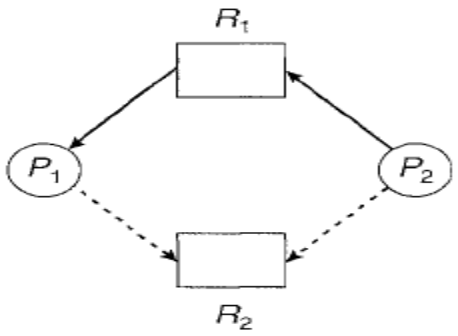

Figure 7.6 Resource-allocation graph for deadlock avoidance.

We introduce a new type edge called a claim edge.

A claim edge P_i->R_j indicates that process Pi may request resource R_j at some time in the future.

This edge similar to a request edge in direction but is represented in the graph by a dashed line.

The claim edge P_i-> R_j is converted to a request edge.

Similarly , when a resource R_j is released by P_i , the assignment edge

R_j->P_i is reconverted to claim edge P_i->R_j.

We note that the resources must be claimed a priori in the system.

Before process Pi starts executing, all its claim edges must already appear in the resource allocation graph.

Now suppose that process P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph 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 P_i will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 7.6. Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph (Figure 7.7). A cycle, as mentioned, indicates that the system is in an unsafe state. If P_1 requests R_2, and P_2 requests R_1, then a deadlock will occur.

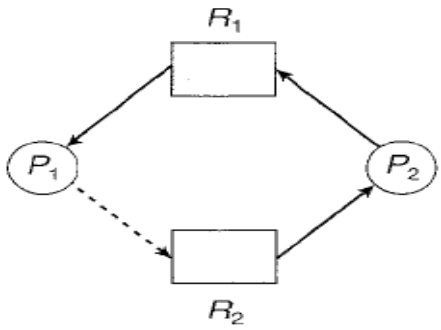

Figure 7.7 An unsafe state in a resource-allocation graph.

Banker's Algorithm

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.

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**. A 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 $Max[i][j] - Allocation[i][i]$.

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$ and $Finish[i] = false$ for $i = 0, 1, \ldots, n - 1$.
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.
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

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 to have allocated the requested resources to process P_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

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.

7.5.3.3 An Illustrative Example

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	

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:

	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.

7.10 Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix *Need*?

b. Is the system in a safe state?

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?

Dead lock 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

7.6.1 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.

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. For example, in Figure 7.8, 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.

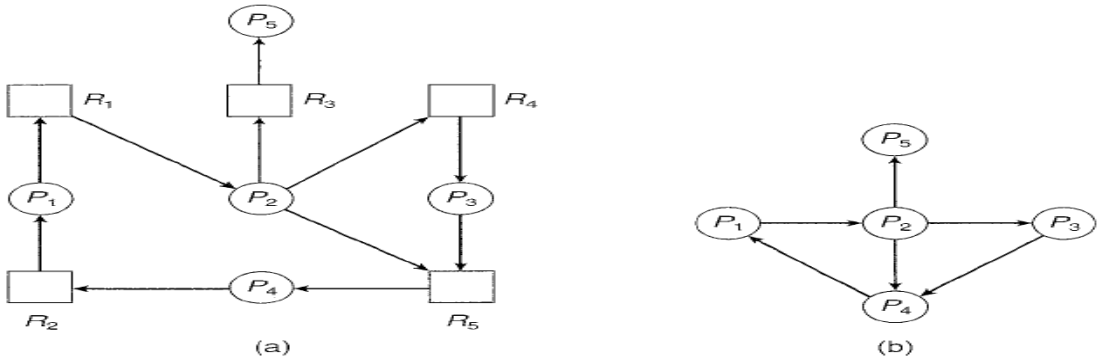

Figure 7.8 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

2. Several Instances of a Resource Type:

1. Data Structures

- **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.

2. Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$. For $i = 0, 1, ..., n\text{-}1$, if $Allocation_i \neq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Request_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == false$ for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, 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.

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:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	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 $<P_0, P_2, P_3, P_1, P_4>$ results in *Finish*[*i*] == *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:

	Request
	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.

7.6.3 Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

4. Recovery from Dead Lock

7.7.1 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.

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

7.7.2 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 during its execution.
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?

7.10 Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix *Need*?

b. Is the system in a safe state?

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?