DEADLOCKS

1.System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. A process must request a resource before using it and must release the resource after using it. 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.

3**. Release**. The process releases the resource.

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release.

To illustrate a deadlock state, 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.

A programmer who is developing multithreaded applications must pay attention to this problem.

2. Deadlock Characterization

Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

Necessary Conditions

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, Pi, ..., Pn }* of waiting processes must exist such that *P-0* 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.

We emphasize that all four conditions must hold for a deadlock to occur.

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 - {Pi, *Pi,,.., Pn,}*, the set consisting of all the active processes in the system, and

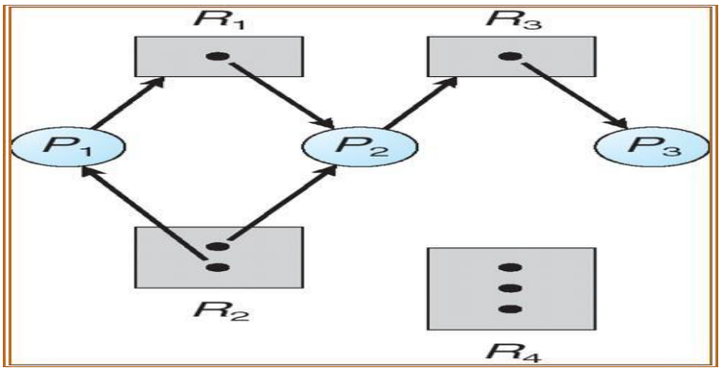
R = {R[, R?, •••/ *Rm},* the set consisting of all resource types in the system.

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

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 *Rj* -* P; is called an **assignment edge.**

Pictorially, we represent each process P, as a circle and each resource type *Ri* as a rectangle. Since resource type *Rj* may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle *R;,* whereas an assignment edge must also designate one of the dots in the rectangle.

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



• The sets P, *R,* and £:

o P={P1,P2,P3}

o R= {R1, *R2,R3,* R4}

o £ = {p1 ->*R1, P2* -> R3/ R,2->*p1, R2* -> P2/ *R2* _> p.,, *R3* ->P3 }

* Resource instances:

o One instance of resource type R1

o Two instances of resource type R2

"' One instance of resource type *R3*

r>Three instances of resource type *R4*

• Process states:

o Process *P1* is holding an instance of resource type R2 and is waiting for an instance of resource type R1.

o Process P2 is holding an instance of *R1* and an instance of R2 and is waiting for an instance of *R3.*

o Process P3 is holding an instance of R3.

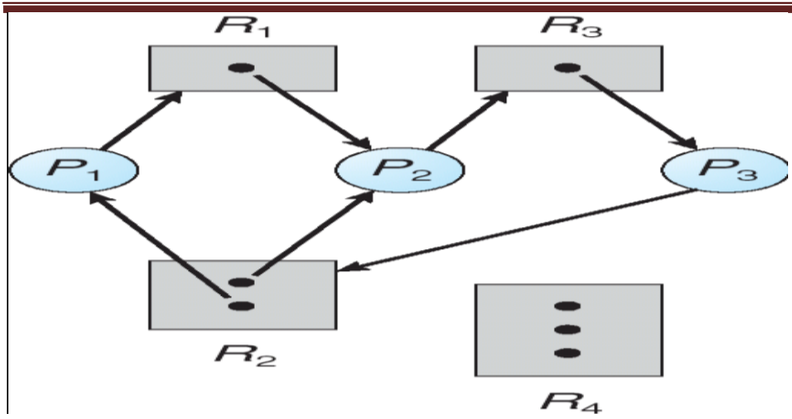
Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.

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.

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



Resource-allocation graph with a deadlock.

P1 ---+ *R* 1 ---+ P2 ---+ R3 ---+ *P3* ---+ R2 ---+ P1

P2 ---+ R3 ---+ *P3* ---+ R2 ---+ P2

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

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 LINUX and Windows.

3. Deadlock Prevention

As we noted ,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 for nonsharable resources. A process never needs to wait for a sharable resource.So, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable,

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

. An alternative protocol allows a process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we 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 copying the disk file to the printer, it releases these two resources and terminates.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption

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 currently being held are preempted.

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.

To illustrate, we let *R = {R1, Ri, ..., Rm}* be the set of resource types. We 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.

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type— say, Rj. After that, the process can request instances of resource type R; if and only if F(R;) > F(R,). If several instances of the same resource type are needed, a *single* request for all of them must be issued. Alternatively, we can require that, whenever a process requests an instance of resource type R,, it has released any resources *R.* such that *F{Rj) >F(Rj).*

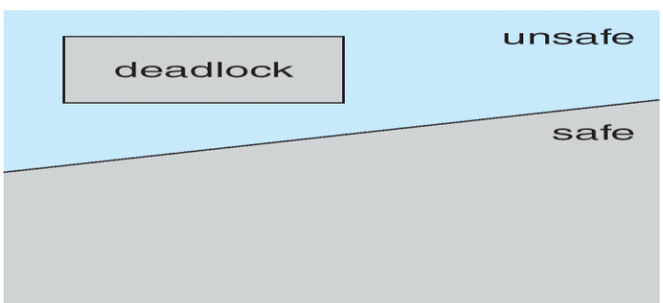
If these two protocols are used, then the circular-wait condition cannot hold.

4. Deadlock Avoidance

Possible side effects of preventing deadlocks are low device utilization and reduced system throughput. An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. With this 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. To making this decision the system 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 *<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 no such sequence exists, then the system state is said to be *unsafe.*



Safe, unsafe, and deadlock state spaces.

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

To illustrate, we 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.

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

At time To, 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 5 available tape drives); then process P0 can get all its tape drives and return them (the system will then have 10 available tape drives); and finally process P2 can get all its tape drives and return them

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.

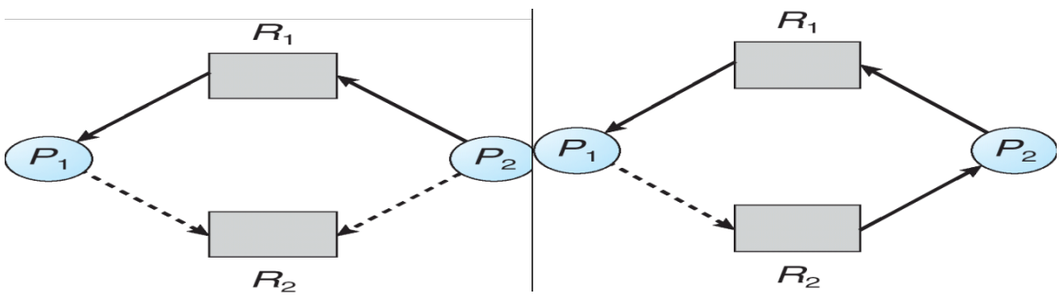
Resource-Allocation-Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, resource-allocation graph can be used for deadlock avoidance. 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. Similarly, when a resource *Rj* is released by *Pi,* the assignment edge *Rj* -» P,- is reconverted to a claim edge *P; —> Rj.* Suppose that process P, requests resource *Rj.* The request can be granted only if converting the request edge P, —» *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 inan unsafe state. Therefore, process *P:* will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph. Suppose that *P1,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.



Resource-allocation graph for deadlock avoidance. An unsafe state in a resource-allocation graph.

Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The *banker's algorithm* algorithm is applicable to such a system. 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.

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. Then a 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. 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 *Availab!c[f]* equals *k,* there are *k* instances of resource type *Ri* 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 Rj.

• **Allocation**. An *n* x mmatrix 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* 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 *Rj* to complete its task. Note that *Need*[i][/] equals *Max[i] [j] - Allocntion[i] [j]*.

To simplify the presentation of the banker's algorithm, we next use some notation. Let X and Y be vectors of length *n.* We say that X < Y if and only if *X[i]* < Y[i] for all i = 1, 2, ..., *n.* For example, if x= (1,7,3,2) and Y =(0,3,2,1), then Y < X. We can treat each row in the matrices *Allocation* and *Need* as vectors and refer to them as *Allocationi* and *Needi.* The vector *Allocationi,* specifies the resources currently allocated to process Pi; the vector *Needi* specifies the additional resources that process Pi may still request to complete its task.

Safety Algorithm

We can use this 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 = A v a i l a b l e a n d *Fiiush[i] = false* for i=0 , 1 , ..., *n - l* .

2. Find an i such that both

a. *Finish[i] ==false*

b. *Neei <= Work*

If no such i exists, go to step 4.

3. *Work = Work + Allocation,*

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* x n^2operations to determine whether a state is safe.

Resource-Request Algorithm

This algorithm determines if requests can be safely granted or not. Let *Requesti* be the request vector for process Pi.When a request for resources is made by process *Pi,* the following actions are taken:

1. If *Requesti* <= *Needi,* go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If *Requesti* <= *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 - Requesti;*

Allocationi, = Allocationi + Requesti;

Needi = Necdi - Requesti;

If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources. However, if the new state is unsafe, then *Pi* must wait for *Request;*, and the old resource-allocation state is restored.

An Illustrative Example

Finally, to illustrate the use of the banker's algorithm, 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 To, 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, P2, P0*>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 *Request1* = (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 1	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.*

5. 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 provide:

• An algorithm that examines the state of the system to determine whether a deadlock has occurred

• An algorithm to recover from the deadlock

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlockdetection 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 Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs. An edge *Pi* -» Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi —>• *Rqj* and Rq, -» Pj for some resource

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 *n1* operations, where *n* is the number of vertices in the graph.

Several Instances of a Resource Type

The deadlock detection algorithm is applicable to multiple instances of each resource type. 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* x *m* matrix defines the number of resources of each type currently allocated to each process.

• **Request**. An *n* x *in* matrix indicates the current request of each process. If *Request[i] [j]* equals *k,* then process Pi is requesting *k* more instances of resource type *Rj.*

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *in* and *n,* respectively. Initialize *Work - Available.* For *i* = 0 , 1 , . . . , *n-1,* if *Allocation,* ^ 0, then *Finish[i] - false;* otherwise, *Finisli[i] = true.*

2. Find an index *i* such that both

a. *Finish[i] -=false*

b. *Requesti < Work*

If no such / exists, go to step 4.

3. *Work - Work + Allocation!*

Finish[i] = true

Go to step 2.

4. If *Finish[i] == false,* for some /', 0 < / <*n,* then the system is in a deadlocked state. Moreover, if *Finish[i] == false,* then process *P;* is deadlocked.

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

To illustrate this algorithm, we consider a system with five processes *P0* through P4 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 To, we have the following resource-allocation state:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P0	*0 1 0*	*0 0 0*	*0 0 0*
P1	*2 0 0*	*2 0 2*	
P2	*3 0 3*	*0 0 0*	
P3	*2 1 1*	*1 0 0*	
P4	*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 <Pn, *Pi, Pi, P*\, *PA*>results in *Finish[i] -- true* for all *i.*

Suppose now that process *Pj* makes one additional request for an instance of type C. The *Request* matrix is modified as follows:

	Request
	A B C
P0	0 0 0
P1	2 0 1
P2	0 0 1
P3	1 0 0
P4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process Po, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes *Pi,* Pi, P3, and P4.

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. 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.

» **Abort all deadlocked processes.** This method clearly will break the deadlock cycle by abort all the dead locked processes, but it increases burden. The system need to restart all the aborted processes.

• **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.

We must determine which deadlocked process (or processes) should be terminated. we should abort those processes whose termination will incur the given factors.

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 type of resources the process has used .
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated?

Resource Preemption

In this, we successively preempt some resources from deadlocked 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.

2. Rollback. If we preempt a resource from a process, We must roll back the process to some safe state and restart it from that state.

3. Starvation. 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 that must be dealt with in any practical system. 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.