Module 3

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**.

System Model

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires carrying out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system.

We shall analyze deadlocks with the following assumptions:

- A process must request a resource before using it. It must release the resource after using it. (Request --use -- release)
- A process cannot request a number more than the total number of resources available in the system.

For the resources of the system, a resource table will be kept, System Table, which 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.

Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources.

Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

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_n} of waiting processes must exist such that Po is waiting for a resource held by P_1, P_1 is waiting for a resource held by P2 , ... , P_{n-1} and Pn is waiting for a resource held by Po.

Resource Allocation Graph

Deadlocks can be described precisely with a DIRECTED GRAPH called 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 R; is denoted by Pi -> R_j; 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 R_j; →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 (as shown in figure 3.1)

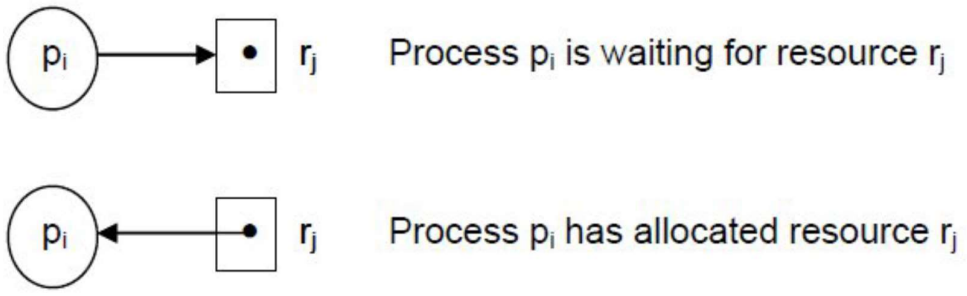

Figure 3.1 Request edge (above) and assignment edge (below).

Another example of RAG is as shown in figure 3.2.

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.

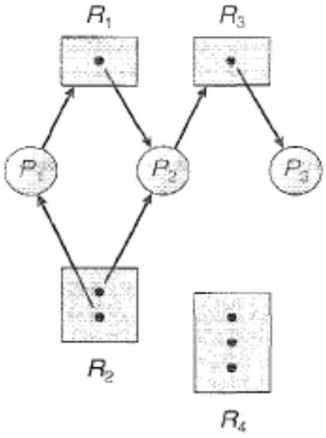

Figure 3.2 Resource allocation graph.

Deadlock Detection using RAG:

1. If the resource allocation graph contains no cycles, then there is no deadlock in the system at that instance.

2. If the resource allocation graph contains a cycle, then a deadlock may exist.

3. If there is a cycle, and the cycle involves only resources which have a single instance, then a deadlock has occurred

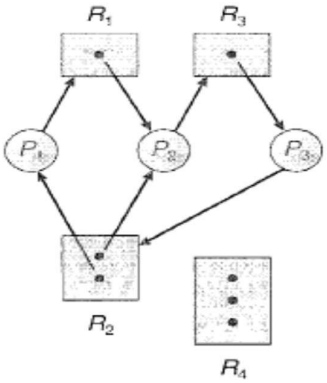

Figure 3.3 Resource allocation graph with a deadlock.

In figure 3.3 Processes P_1, P_2, and P_3: are deadlocked. Process P_2 is waiting for the resource R_3, which is held by process P_3. Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2. In addition, process P_1 is waiting for process P_2 to release resource R_1.

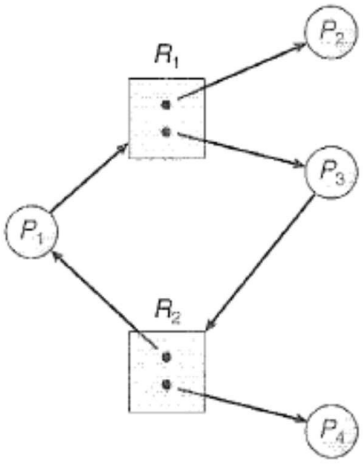

Figure 3.4 Resource allocation graph with a cycle but no deadlock.

Figure 3.4 shows how there is no possibility of deadlock even if the cycle is present. 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.

Methods for Handling Deadlocks

The deadlock problem can be solved in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
- We can allow the system to enter a deadlocked 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; it is then up to the application developer to write programs that handle deadlocks.

To ensure that deadlocks never occur, the system can use either a **deadlock prevention** or a deadlock-avoidance scheme. Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold.

Deadlock avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Deadlock Prevention

By ensuring that at least one of these (Mutual Exclusion, Hold and Wait, No preemption, Circular Wait) conditions cannot hold, we can prevent the occurrence of a deadlock.

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.

2. **Hold and Wait**

To ensure th.at 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.

3. **No Preemption**

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

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.

Deadlock Avoidance

Deadlock-prevention algorithms prevent deadlocks by restraining how requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock cannot occur and hence, that deadlocks cannot hold. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

The various algorithms that use 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 a deadlocked state. Such an algorithm defines the deadlock-avoidance approach.

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 $< P_1, P_2, ..., P_n >$ is a safe sequence for the current allocation state it 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 $j < i$.

In this situation, 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 +1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

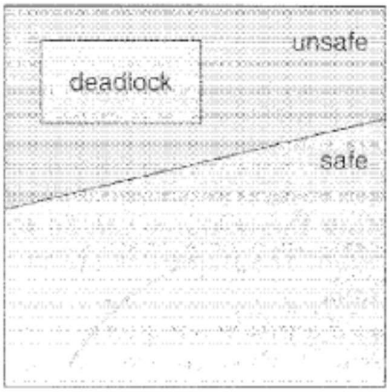

Figure 3.5 Safe, unsafe, and deadlock state spaces.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 3.5). An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent processes from requesting resources such that a deadlock occurs: The behavior of the processes controls unsafe states.

Consider a system (as shown in figure 3.6) with 12 magnetic tape drives and 3 processes:P_0, P_1, and P_2.Process P_0 requires 10 tape drives, process P_1 may need 4 tape drives, and process P_2 requires 9 tape drives. Suppose that, at time t_0, process P_0 is holding 5 tape drives, process P_1 is holding 2 tape drives, and process P_2 is holding 2 tape drives. (Thus, there are 3 free tape drives.)

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

Figure 3.6 Maximum and current need of the processes.

At time t_0, the system is in a safe state. The sequence < P_1, P_0, P_2 > satisfies the safety condition. Process P_1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives); then process Po can get all its tape drives and return them (the system will then have 10 available tape drives); and finally process P_2 can get all its tape drives and return then (the system will then have all 12 tape drives available).

Resource-Allocation-Graph Algorithm

In 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. Similarly, when a resource Rj is released by Pi the assignment edge Rj→Pi is reconverted to a claim edge Pi→Rj.

Suppose 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. 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.

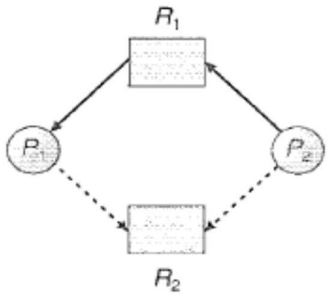

Figure 3.7 Resource-allocation graph for deadlock avoidance.

To illustrate this algorithm, we consider the resource-allocation graph of Figure 3.7. 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 3.8). A cycle 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.

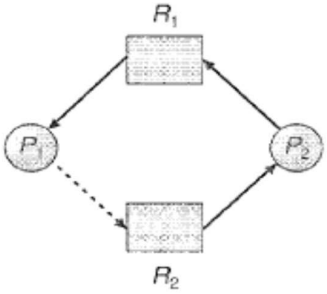

Figure 3.8 An unsafe state in a resource-allocation graph.

Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. 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.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several **data structures** must be maintained to implement the banker's algorithm:

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 Rj.
- **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 Rj .
- **Need**. An n x m matrix indicates the remaining resource need of each process. If Need[i][j] equals k then process p) may need k more instances of resource type Rj to complete its task. Note that Need[i][j] equals Max[i][j] - Allocation[i][j].

These data structures vary over time in both size and value.

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

Resource- Request Algorithm

Resource-Request algorithm determines whether requests can be safely granted or not. Let $Request_i$ be the request vector for process Pi If $Request_i[j]$ == k, then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, 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, 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- $Request_i$

$Allocation_i = Allocation_i + Request_i$

$Need_i = Need_i - Request_i$

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 $Request_i$; 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 P_0 through P_4 and three resource types A, B, and C. Resource type A has 10 instances (as shown in figure 3.9) , resource type B has 5 instances, and resource type C has 7 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	

Figure 3.9 Snapshots of processes at time t_0.

The content of the matrix Need is defined to be Max -Allocation and is shown below in figure 3.10:

	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

Figure 3.10 Need matrix of different processes.

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 PI 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 $Request_i \leq$ 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 as shown in figure 3.11

	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	

Figure 3.11 New state after allocation.

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.

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 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.

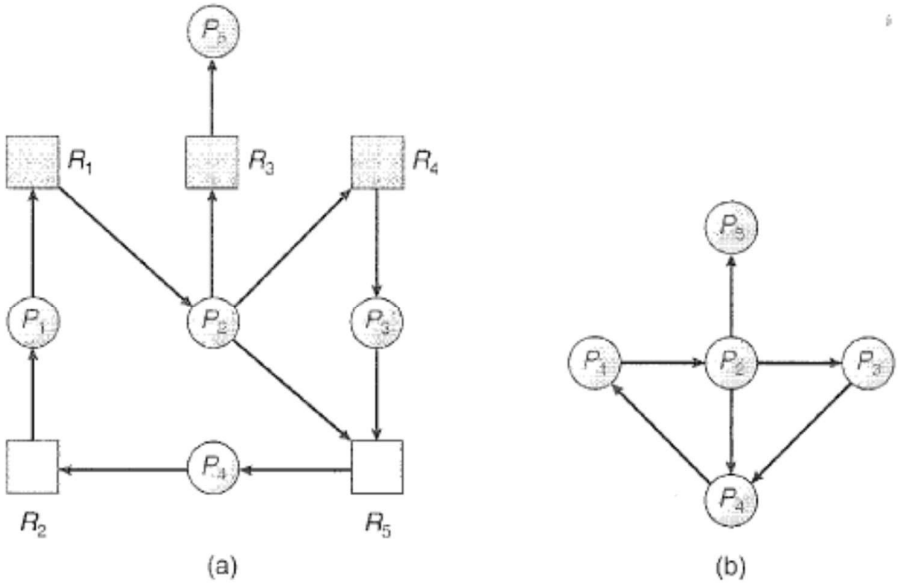

Figure 3.12 (a) Resource allocation graph (b) Corresponding wait-for graph

As shown in figure 3.12, 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.

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.

Data structures used are:

- **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 m 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 .

Algorithm

1. Let Work and Finish be vectors of length m and n respectively. Initialize Work= Available. For i=0,1,. . . ,n-1, if Allocationi ≠ 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≤i≤n, then the system is in the deadlocked state. Moreover, if Finish[i] = false, then process Pi is deadlocked.

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

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?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting

processes. In the extreme, we can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the deadlocked set of processes but also the specific process that "caused" the deadlock.

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.

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.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. 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 type 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?

Resource Preemption

To eliminate deadlocks, we successively preempt some resources from the system and give these resources to other processes until the deadlock cycle is broken. If preemption is required, then the following three factors to be considered.

1. Selecting a victim: Which processes and which resources are to be preempted. The cost of preemption should minimize the cost

2. Rollback: If we preempt a resource from a process, what should be done with that process? We must roll back the process to some safe state and restart it from that state

3. Starvation: How can we guarantee that the resources will not always preempt from the same process?

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.

Module 4

Memory management strategies

Background

Memory is central to the operation of a modern computer system. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses. A typical instruction-execution cycle first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory.

Basic Hardware

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. Any instructions in execution, and any data being used by the