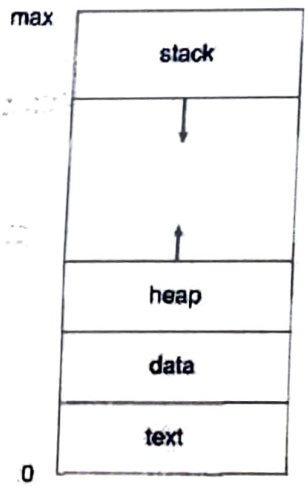
UNIT-3

Process

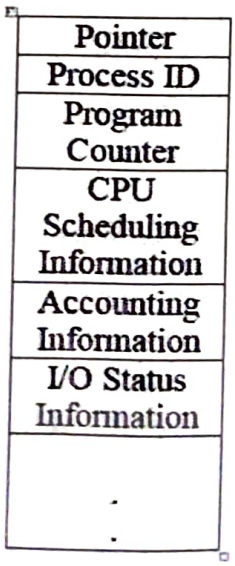
- Program in execution is called process.
- Process is active entity while program is passive entity.
- Process is smallest unit of work individually scheduled by operating system.



Process in memory.

Process Control Block/PCB

- PCB holds all the information needed to keep track of a process.
- It is a data structure maintained by Operating System.
- OS creates PCB for every process.
- It is useful in multi programming environment.

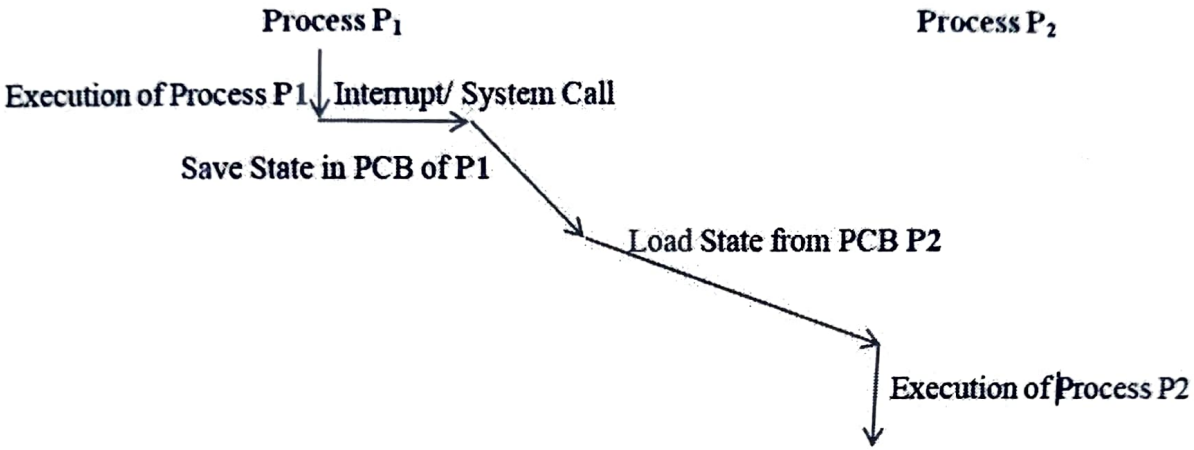


Where:

- ✓ Pointer holds address of parent process
- ✓ Process ID holds unique identification number for process.

- ✓ Program counter holds address of next instructions to be executed.
- ✓ CPU scheduling information holds information like priority of process.
- ✓ Accounting Information holds the information of amount of CPU used.
- ✓ I/O status information contains the information of I/O devices allocated to process.

Process State switching Diagram



- ❖ Process P1 execution starts in user mode when it needs an event to occur like I/O Request, the process sends an interrupt to CPU, CPU stores the state of process P1 in its PCB and load state of P2 from its PCB. After loading state of P2,execution of process P2 begins.
- ❖ Execution of Process occurs in User Mode while Save and Load of PCB occurs in Kernel Mode.

Schedulers

Long Term Scheduler/Job Scheduler

- It selects process from Job Queue and assigns it to Ready Queue.
- It changes state of processes from New State to Ready State.

Short Term Scheduler/CPU Scheduler

- It selects process from Ready Queue and assigns it to CPU.
- It changes state of processes from Ready State to Running State.
- Dispatcher is responsible for saving the context of one process (i.e. content of PCB) and loading the context of another process.

Medium Term Scheduler

- It removes the processes from Main Memory.
- It reduces degree of multiprogramming.
- It is responsible for swapped out process.

Dispatcher: Dispatcher is part of Short Term Scheduler that performs following functions.

- Context switching
- Switching to User Mode
- Jumping to the proper location in the user program to restart that program

Queues used in Process Scheduling

Job Queue: It contains all processes of system.

Ready Queue: It contains all the processes that reside in Main Memory and ready to be executed.

Waiting Queue: It contains all the processes that are waiting for I/O.

Note: Job Queue, Ready Queue and waiting Queue all are implemented using Linked List.

Life Cycle of Process

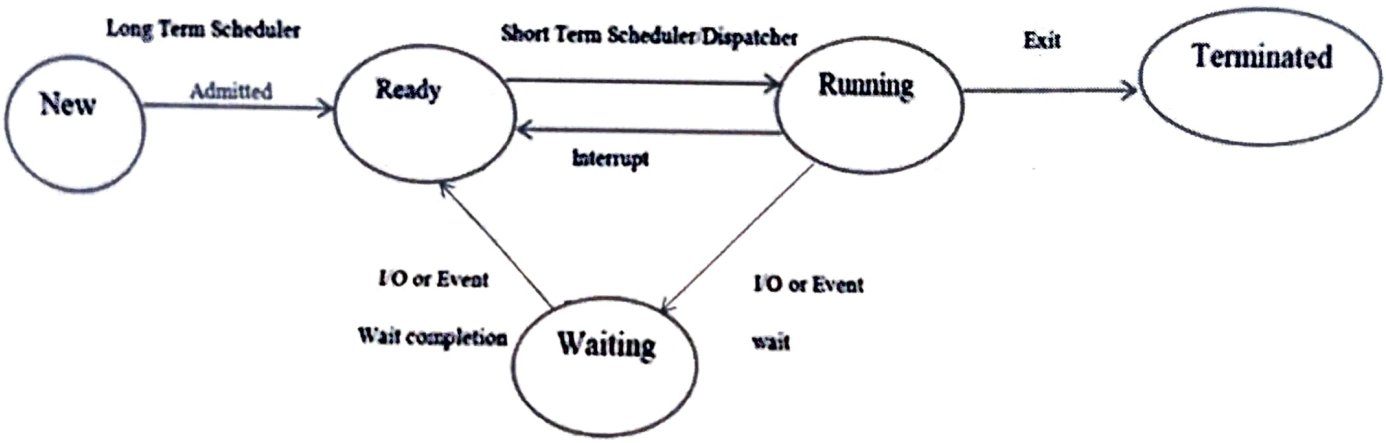
New State: When a process is created it is in new state. In new state process resides in Job Queue (Secondary Memory).

Ready State: State of process is called Ready State when Process resides in Ready Queue (Primary Memory) and waiting for CPU.

Running State: State of process is called Running state if CPU is assigned to process.

Waiting State: State of process is called Waiting State when Process resides in waiting Queue (Primary Memory) and waiting for some event like I/O to occur.

Terminated State: State of process is called Terminated State when the Process has completed its execution successfully.



CPU SCHEDULING

Removal of the running process from the CPU and selection of another process on the basis of particular strategy is called CPU Scheduling.

Non Preemptive Vs Preemptive

Non Preemptive Scheduling or [redacted]: In non-preemptive scheduling, once the CPU is allocated to a process, keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state i.e. when a process reaches to running state it can either switches to waiting state or terminates.

Preemptive Scheduling: Preemptive scheduling allows a process to switch from running state to ready state or waiting state to ready state. In Preemptive scheduling, once the CPU is allocated to a process, process can release the CPU before terminating the process.

Scheduling Performance Criteria

Criteria that are made for comparing scheduling algorithms are given below:

CPU Utilization: We want to keep the CPU as busy as possible. Conceptually CPU utilization range from 0 to 100 percent while in real system it should range from 40 to 90 percent.

Throughput: Number of processes that can completed per time Unit is called Throughput.

Turnaround Time: The interval from time of submission of a process to time of completion is called Turnaround Time of that process.

Waiting Time: Sum of periods spent waiting by a process in the ready queue is called waiting time of the process.

Response Time: Time interval from submission of a request until the first response is produced is called Response Time.

NOTE: It is desirable to:

- ✓ maximize CPU Utilization and Throughput
- ✓ minimize Response Time, Waiting Time and Turnaround Time

Scheduling Algo.

Pre-emptive	Non-pre emptive
SRTF	FCFS (First come first Serve)
LRTF	SJF (shortest job first)
Round Robin	LJF (longest job first)
Priority based	HRRN (Highest Response Ratio next)
	→ Multilevel Queue.

Arrival Time → The time at which Process enter the Ready Queue

Burst time → Time req. by a process to execute on CPU.

Completion time → The time at which process complete its execution

Turn Around time → {Completion time − Arrial time}

Waiting time → {Turn Around time − Burst time}

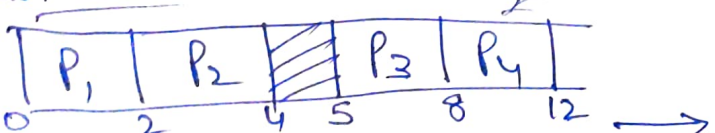
Response time → {(Time at which a process get CPU first time) − (Arrival time)}

FCFS (In Non Preemptive) (RT same as WT)

(CT − AT) (TAT − BT)

Process No.	Arrival time	Burst time	Completion time	TAT	WT	RT	
P_1	0	2	2	2	0	0	(0-0)
P_2	1	2	4	3	1	1	(2-1)
P_3	5	3	8	3	0	0	(5-5)
P_4	6	4	12	6	2	2	(8-6) ↑ CPU

Gantt chart



Avg. TAT = $\frac{14}{4}$

Types of Scheduling Algorithm

First Come First Served Scheduling (FCFS)
Shortest Job First Scheduling (SJF)
Priority Scheduling
Round Robin
Multilevel Queue Scheduling
Multilevel Feedback Queue Scheduling

FCFS or First Come First Served Scheduling

- FCFS Scheduling is non Preemptive it means in FCFS Scheduling once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU either by terminating or by requesting I/O.
- In FCFS scheduling the process that request the CPU is allocated the CPU first.

Advantages:

- Easy to implement
- Easy to understand

Disadvantages:

- Average waiting time in FCFS Scheduling is often quite longer.
- Sometimes convey effect may occur in FCFS Scheduling.

Example: .....

Shortest Job First (Preemptive and Non-Preemptive)/ Shortest Remaining Time First(Preemptive) / Shortest Next CPU Burst Scheduling(Preemptive)

- SJF scheduling can be either preemptive or non-preemptive.
- In SJF CPU is assigned to the process that has the smallest next CPU Burst.
- If the next CPU burst of two processes is the same FCFS Scheduling is used to break the tie.
- This approach gives minimum average waiting time for a given set of processes.

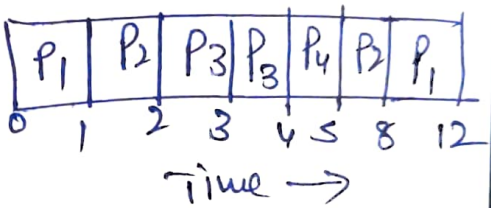
Priority Scheduling

- In this scheduling a priority is associated with each process and CPU is allocated to the process with the highest priority.
- Equal priority processes are scheduled in FCFS order.
- SJF scheduling is special case of priority scheduling where the priority is inverse of the next CPU burst.
- Priority scheduling can be either preemptive or non-preemptive.
- Major problem with Priority Scheduling is problem of starvation.
- Solution of the problem starvation is aging, where aging is a technique of gradually increasing the priority of the processes that wait in the system from long time.

Example: .....

Higher the no. higher the priority.

Priority	Process	AT	BT	CT	TAT	WT
10	P1	0	5	12	12	7
20	P2	1	4	8	7	3
30	P3	2	2	4	2	0
40	P4	4	1	5	1	0



Round Robin Scheduling

- *It is designed especially for time sharing system or multi-tasking system.*
- It is similar to FCFS scheduling but preemption is added to enable the system to switch between processes.
- In this technique ready queue is treated as circular queue. CPU scheduler goes around the ready queue allocating the CPU to each process for a time interval up to 1 time-quantum (or time-slice).
- Round Robin scheduling is preemptive
- This approach gives minimum average response time for a given set of processes.

Example: RR (Given Time Quantum → 2)

Process	AT	BT	CT	TAT	WT	RT
P1	0	5, 3, 1				
P2	1	4, 2				
P3	2	2, 0				
P4	4	1				

Gantt chart

Ready Queue: P1 P2 P3 P4 P1 P2 P1

Running Queue: P1 P2 P3 P1 P4 P2 P1

0 2 4 6 8 9 12 12

context Switching.

if two or more processes are waiting on happening of some event which never happens, then we say these processes are involved in deadlock then that state is called deadlock.

Deadlock

A → ← B

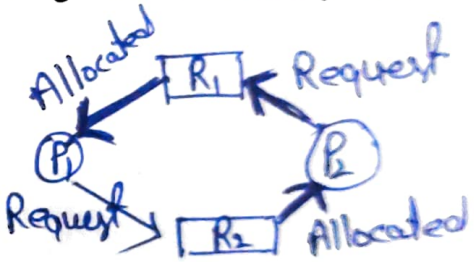
In a multiprogramming environment, several processes may compete for a finite number of resources. A process request resources: if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources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

System Model: A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. A process must request a resource before using it and must release the resource after using it.

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.

Use: The process can operate on the resource.

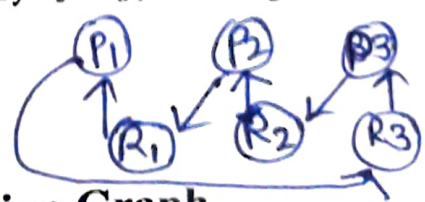
Release: The process releases the resource.



(Hold & wait)

Necessary conditions for deadlock

1. **Mutual Exclusion:** At least one resource must be held in a non sharable mode i.e. only one process at a time can use the resource. If another process requests the resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait:** A process must be holding one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No Preemption:** Resources cannot be preempted i.e. a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait**: A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exists such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by $P_2 \ldots P_{n-1}$ is waiting for a resource held by P_n and P_n is waiting for a resource held by P_0.



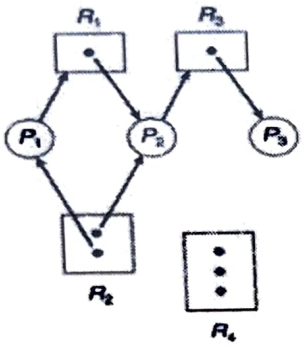
Resource allocation Graph

In Resource allocation graph we represent--

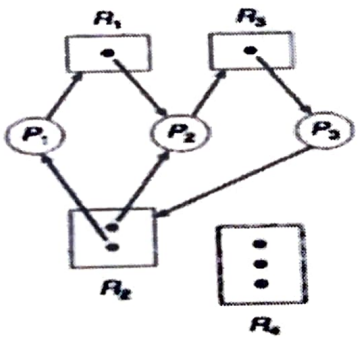
- Each process P_i as circle and each resource type R_j as a rectangle. Each instance of resource type R_j is represented by dot within the rectangle.
- A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$ If process P_i has requested an instance of resource type R_j.

- A directed edge from process R_j to resource type P_i is denoted by $R_j \rightarrow P_i$ If an instance of resource type R_j has been allocated to process P_i.
- 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.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.

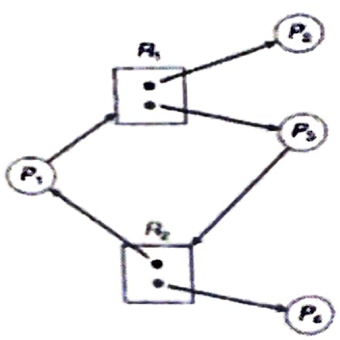
Example1: No cycle with No Deadlock



Example 2: Cycle with Deadlock



Example 3: Cycle with No Deadlock



Methods for handling Deadlocks

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 deadlocked state (Deadlock Prevention or Avoidance).
- We can allow the system to enter a deadlocked state, detect it, and recover (Detect and resolve).
- We can ignore the problem altogether and pretend that deadlocks never occur in the system (Ignorance).
    - ✓ To ensure that deadlocks never occur, deadlock prevention scheme, provides a set of methods for ensuring that at least one of the necessary conditions cannot hold.
    - ✓ To ensure that deadlocks never occur, deadlock avoidance scheme, 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

To ensure that deadlocks never occur, deadlock prevention scheme, provides a set of methods for ensuring that at least one of the necessary conditions cannot hold.

Mutual Exclusion

The mutual-exclusion condition must hold for nonsharable 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. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, 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 to request resources only when it has none. 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.

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 then all resources the process is currently holding are preempted. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait

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. 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 two resources and to determine whether one precedes another in our ordering.

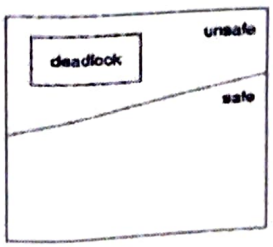
Deadlock Avoidance

Deadlock avoidance require additional information about how resources are to be requested. We can use one of the following two methods for deadlock avoidance.

1. Resource allocation graph algorithm
2. Banker's Algorithm.

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. i.e. a system is in a safe state only if there exists a safe sequence.
- ❖ If safe sequence not exists, then the system state is said to be unsafe.
- ❖ A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks.



Banker's Algmorithm

- Banker's algorithm is also applicable to a resource allocation system with **multiple instances** of each resource type.
- It is is **less efficient** than the resource-allocation graph scheme.
- In this scheme when a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This nun1.ber 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.

Data Structure'smaintained for implementation of Banker's Algorithm: Let n is the number of the processes in the system. and m is the number of resource type.

Available. A vector of length m indicates the number of available resources of each type.

Max. An n x m matrix defines the maximum demand of each process.

Allocation. An 11 x m matrix defines the number of resources of each type currently allocated to each process

Need. An n x m matrix indicates the remaining resource need of each process. Need[i][j] = Max[i][j] - Allocation [i][j]

Safety Algorithm: It is used for finding out whether or not a system is in a safe state.

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

2- Find an 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.

Resource-Request Algorithm: It is used for determining whether requests can be safely granted. Let Request; be the request vector for process P;. If Request; [j] == k, then process P; wants k instances of resource type Rj. When a request for resources is made by process P;, 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; must wait, since the resources are not available.

3- Have the system pretend to have allocated the requested resources to process P; by modifyil1.g 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; is allocated its resources. However, if the new state is unsafe, then P; must wait for Request;, and the old resource-allocation state is restored

Example: 1(a) Consider a system with five processes Po through P4 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 T0, the following snapshot of the system has been taken:

	Allocation	**Max**
	ABC	ABC
P0	010	753
P1	200	322
P2	302	902
P3	211	222
P4	002	433

since available = Total – Total Allocation

$$= [10\ 5\ 7] - [\ 725] = [325]$$

Need matrix is defined as Need = Max- allocation

	Allocation	Max	Need	Available
	ABC	ABC	ABC	ABC
P0	010	753	743	332
P1	200	322	122	
P2	302	902	600	
P3	211	222	011	
P4	002	433	431	
Total	725			

Select Process	Available/ Work=Available+ Allocation of selected process
P1	Available=[332]+[200]= [532]
P3	Available=[532]+[211]= [743]
P4	Available=[743]+[002]= [745]
P2	Available=[745]+[302]= [1047]
P0	Available=[1047]+[010]= [1057]

We can find a safe sequence **< Pl, P3, P4, P2, Po>** so system is in safe state.

1(b) 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 $Request_1 \leq$ 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	Max	Need	Available
	ABC	ABC	ABC	ABC
P0	010	753	743	332-102=**230**
P1	200+102=**302**	322	122-102=**020**	
P2	302	902	600	
P3	211	222	011	
P4	002	433	431	
Total	725			

we execute our safety algorithm and find that the sequence **<P1, P3, P4, Po, P2>** satisfies the safety requirement. Hence, we can immediately grant the request of process P1.

Recovery From Deadlock:

a) **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.
   I. **Abort all deadlocked processes**: This method clearly will break the deadlock cycle, but at great expense.
   II. **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 algorithnc rnust be invoked to determine whether any processes are still deadlocked.

b) **Resource Preemption:** To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes 1-m til the deadlock cycle is broken.