**II Unit**

**CPU Scheduling**

**CPU scheduling algorithm properties**

CPU scheduling is nothing but deciding which process in ready queue should be given to CPU.A best CPU scheduling algorithm should satisfy the following properties.

1. **CPU Utilization:**

Since CPU is costly device, we want to keep it as busy as possible, it ranges from 0% to 100% if CPU utilization is 80% then it means 80% of the time CPU is busy & 20% of time CPU is idle, if CPU utilization is more, then the algorithm is considered to be efficient.

1. **Throughput:**

It is nothing but no. of Jobs executed in given time interval. If more no. of Jobs can be executed in given time interval then algorithm is considered to be efficient.

1. **Turnaround Time:**

TAT = completion time - Submission time

= waiting time + execution time.

Where completion time is the time at which program completes execution. Submission is the time at which program is entered into mm into system for any algorithm the turnaround time of a job should be minimum.

1. **Waiting time:**

Often turn around time may not be a good measure because TAT depends on waiting time of the job waiting time is defined as the amount of time a job waits in ready queue for getting selected by CPU + the amount of time a job waits in device a queue for getting selected by I/O device for an efficient algorithm the waiting time should be minimum.

**CPU Scheduling Algorithms**

1. **FCFS (First Come First Served)**

It is a simplest CPU scheduling algorithm to implements. It was implemented on non-time sharing systems. Here the process which request for CPU first is allocated CPU first FCFS is implemented with a FIFO queue. When a process enters into Ready queue its PCB is appended at end of Ready queue. When the CPU is free, the STS allocate the PCB at the front of queue to CPU and the running process is removed from the ready queue

**Ex:**Consider three jobs J1,J2,J3

|  |  |  |
| --- | --- | --- |
| Job | CPU burst Time | Arrival Time |
| J1 | 20 | 0 |
| J2 | 4 | 0 |
| J3 | 3 | 0 |

The jobs arrived in the order J1, J2, J3 i.e. J1 entered first followed by J2 and lastly J3

|  |  |  |
| --- | --- | --- |
| J1 | J2 | J3 |

**0 20 24 27**

The Turnaround time of Job J1= 20 - 0=20

The Turnaround time of Job J2= 24- 0=24

The Turnaround time of Job J3= 27 - 0=27

Average Turnaround time =



**2. SJF (Shortest Job first)**

SJF associates the length of the CPU burst time with each job. When CPU is free, it is given the job with smallest CPU burst time. Here the job with smallest CPU burst time is called shortest Job and Job with longest CPU burst time is called longest Job. This algorithm executes shortest Job first. If two jobs have same CPU burst time then they are serviced on FCFS basis.

Ex: consider Three jobs as shown below

|  |  |  |
| --- | --- | --- |
| **Job** | **CPU burst Time** | **ArrivalTime** |
| J1 | 20 | 0 |
| J2 | 4 | 0 |
| J3 | 3 | 0 |

|  |  |  |
| --- | --- | --- |
| J3 | J2 | J1 |

0 3 7 27

Here the job with smallest CPU burst is J3. And a job with longest CPU burst is J1.

TAT fir J1 is 27, TAT of J2 is 7, TAT for J3 is 3, Average TAT is =12.3

Thus average turn around time is reduced compared to FCFS and hence it is considered to be an optimal algorithm’) However, implementation wise it is little difficult to implement. Because there is no way to know the CPU burst time of each job in advance. However, OS can make a guess on CPU burst time of each Job by looking at the size of the job (i.e. compiler gives size).

In this algorithm, longer jobs are made to wait for longer time. In busy environment, it may be so happen that longer Jobs may not be executed at all. This is known as starvation.

**3. Priority Algorithm**

Here each job is given some priority based on which CPU is allocated. The Job with highest priority is executed first and job with least priority is executed last. If there are two Jobs with same priority, they are selected on the basis of FCFS. SJF is also one type of priority algorithm. Where priorities are decided on the basis of CPU burst time.

Thus in SJF algorithm **Priority =1/CPU burst time** i.e. lower the CPU burst time higher the priority.

**Ex:**

|  |  |  |
| --- | --- | --- |
| **Job** | **CPU burst Time** | **Arrival Time** |
| J1 | 20 | 0 |
| J2 | 4 | 0 |
| J3 | 3 | 0 |

Here J3 has Highest Priority, since cpu burst time is least. J1 is having least priority since its CPU burst time is highest.

There is one Problem with priority algorithm which is starvation i.e. indefinite waiting or indefinite blocking i.e. A process waits for CPU indefinitely and it is never selected by STS because of its least priority. Thus priority algorithms would leave some lower priority processes waiting indefinitely for CPU and those processes would never be selected. This is overcome by using a technique called aging. This technique will increment the priority of all jobs existing in the system at regular in terms of time.

Ex Let us assume there is a job of priority 1. And priority ranges from 1 to 127. Then aging algorithm will increase the priority of Job by 1 for every 15 minutes (say).Thus the priority of these jobs is gradually increase and finally executed after sometime. Thus aging is a technique to prevent starvation.

1. **Preemptive algorithms:**

A Non - preemptive algorithm is one where in one CPU is given to a job, that Job will not leave CPU until I/O burst takes place or job completed. Thus in non-preemptive algorithm once CPU is given to a job, it cannot be taken out in between the execution.

However, a preemptive algorithm is one where in CPU can be taken out from the without completing its CPU burst.

SJF and priority algorithms can be implemented both in non-preemptive and Preemptive fashion. Whereas FCFs algorithm can be implemented only in non-preemptive fashion. If we try to implement in FCFS in preemptive fashion it behaves like non preemptive FCFS

**4 (a) Preemptive SJF algorithms.**

When CPU is given to a process, it will be with that process until a new Job arrives into a system .once a new Job is arrived, we compare CPU burst time of new Job with the remaining CPU burst time of the current Job. If the CPU burst time of the new Job is less than the remaining CPU burst time of the current job then CPU is preempted form the current Job and is given to the new job. The current Job will be executed. Suppose the remaining CPU burst time of the current Job is less than or equal the CPU burst time of the new Job, then CPU continues execution with current job.

Whereas in non-preemptive SJF algorithm.CPU continues execution with current job irrespective of the CPU burst time of the new jobs. Thus, the main difference between preemptive and non preemptive algorithm is that, in a non –preemptive algorithm the process will not leave CPU, until it is executed where in preemptive algorithm the CPU leaves the process even if it is allocated

Consider four jobs as shown below

|  |  |  |
| --- | --- | --- |
| **Job** | **CPU burst Time** | **Arrival Time** |
| J1 | 10 | 0 |
| J2 | 5 | 1 |
| J3 | 4 | 2 |
| J4 | 2 | 3 |

These jobs are executed in the following order in non preemptive SJF algorithm

**Non-preemptive SJF**

|  |  |  |  |
| --- | --- | --- | --- |
| J1 | J4 | J3 | J2 |

0 10 12 16 21

TAT for J1=10-0 TAT for J2=21-1=20

TAT for J3=16-2=14 TAT for J4=12-3=9

Avg TAT =10+20+14+9/4= 13.3

**Preemptive SJF:** Arrival timings are different

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| J1 | J2 | J3 | J4 | J3 | J2 | J1 |

0 1 2 3 5 8 12 27

TAT of J1=21-0=21 TAT for J2=8-1=7

TAT of J3=12-2=10 TAT of J4=5-3=2

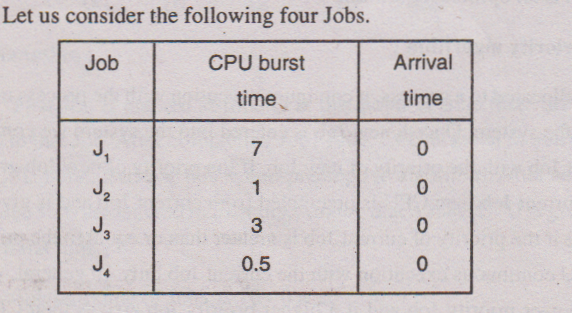
Avg TAT=



1. **Round Robin Algorithm**

Round Robin scheduling algorithm is designed especially for time sharing systems. Here CPU is given to each Job for a specific time period called time quantum or time slice. The time quantum ranges from 10m sec to 100 msec. The Round Robin algorithm is implemented with FIFO circular queue. The circular queue contains Jobs waiting for CPU. New process is inserted at the rear end of the circular queue.

STS selects a Job from Ready queue and gives CPU to that Job, before CPU is given, a timer is set. Once the time quantum is completed. The timer sends an interrupt to CPU and CPU is released form that Job. Now this Job is put at the end of Ready queue for the remaining CPU burst. Suppose CPU burst time of a Job is less than the time quantum, the Job itself will release CPU without getting interrupt. Once CPU is released from a Job, STS picks the next Job form Ready queue and gives CPU to it.



Let us assume Jobs arrived in the order of J1, J2, J3 and J4. Let the time quantum be one see. Then Jobs are executed as follows.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| J1 | J2 | J3 | J4 | J1 | J3 | J1 | J3 | J1 | J1 | J1 |

0 1 2 3 3.5 4.5 5.5 6.5 7.5 8.5 9.5 11.5

TAT of J1=11.5-0=11.5, TAT of J2=2-0=2 TAT of J3=7.5-0=7.5 ,

TAT of J1=3.5-0=3.5 AVG TAT=6.125

As the time quantum is increasing, Average TAT is also increases. Hence lower the time quantum, optimal the algorithm

RR is a preemptive algorithm. If we put time quantum maximum, then it have non- preemptive .The performance of RR algorithm depends on time quantum i.e. If time quantum is low, then Jobs will be wait in the Ready queue for minimum time. Time quantum is more Jobs will wait in the Ready queue for a long time. Thus as a time quantum is increasing, the waiting time for each Job increases. At one point of time RR behaves like FCFS when time quantum is too large. If the time quantum is too small, R.R is called processor sharing because each user feels that he has his own CPU. But actually in the system there is only one CPU.

**Context Switch**

In Round Robin algorithm CPU gets an interrupt from timer at the end of time quantum. Now operating system must save the details of the execution done in this time quantum period in PCB. When CPU is again given to the same Job after some time, the contents of PCB must be retrieved and loaded in appropriate registers. This is known as context switch. Thus context switch is nothing but loading the contents of registers in PCB at the end of the time quantum for a Job and retrieving the contents of PCB and loading in registers at the beginning of the next time quantum. The time required for context switch is known as context switch time. This time varies from machine to machine. Since the number of registers present in the computer.

Let us assume a Job requires 12 units of CPU time. Let the time quantum be 2 units of time. Then the Job will be executed in 6 intervals. The number of context switches is 5. Since each context switch will take some time. The execution of the process will slow down as the number of context switches increase.

**Multi Level queues:**

Here no. of queues is maintained based on the priority of the jobs. Let us assume priorities are ranging from 0to 3. Then we require 4 queues to store these jobs.

Queue 0 contains jobs with highest priority i.e. 3.

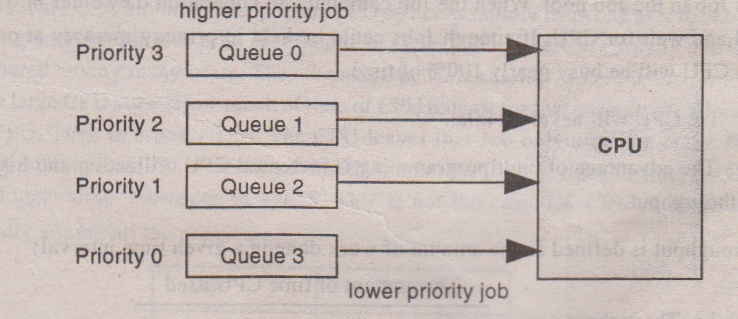
Q1 contains jobs with highest priority i.e. 2

Q2. Contains Jobs with next higher priority i.e. 1

Q3. Contains Jobs with next higher priority i.e. 0

Thus the Jobs with same priority are placed in a same queue. Whenever CPU is free, STS has to select Jobs with highest priority i.e. from queue ‘0’. Thus in this method; jobs with highest priority are serviced first followed Jobs with lowest priority. Jobs are selected from a queue using any one of the algorithms such as FCFS. SJF. RR etc. Jobs in queue 3 are selected only when Q0, Q1, Q2 are empty. Jobs in Q2 are selected only when Q0 and Q1 are empty. Jobs in Q1 are selected when Q0 empty. If Q0 contains Job then any one of them is selected based on the algorithms.

Let the algorithm followed be preemptive then when CPU is servicing a lower priority Job from Q3 and a higher priority Job enters into QO. Then CPU will preempt (suspend) the lower priority Job and starts execution with higher priority Job. However in non-preemptive algorithm, CPU continues execution with lower priority Job only i.e. It will not respond to higher priority Job until I/O burst is encountered for lower priority Job.



The Jobs in Q3 will be starvated i.e. they have to wait for CPU for a long time. To overcome this we use aging algorithm. Here the priority of lowest priority Job is incremented by 1 at regular intervals of time. So the Job goes to the upper queue. Each time its priority is incremented by I after some time Job goes to highest priority queue and executed.

**Multilevel Feedback Queues**

In multilevel queues, Jobs are permanently assigned to a queue upon entry to the system i.e. Jobs do not move between the queues (assuming aging algorithm is not employed). Where as in multilevel feedback queues Jobs can be moved between the queues depending on the CPU burst time, waiting time, I/O burst time etc.

Ex: Consider 3 queues Q0, Q1, Q2. Let us assume. Jobs are moved between these queues depending on CPU burst time. Any Job which enters into system is put in Q0. Where each Job is given a time quantum of 8msec. If it does hot complete within the time, it is moved to Q1. Thus Jobs with CPU burst time 8m sec are executed and are not moved to QI. Since they complete execution from Q0 itself.

However, if a Job CPU burst time is > 8 then it is moved to Q1. Jobs are selected form Q1 only when QO is empty. Each Job present in Q1 is given a time quantum of 16m sec. If a Job does not complete within this time, then it is further move to Q2. Thus jobs with CPU burst time between 9 & 24 m sec are executed and are not moved to Q2. Since they complete execution from Q1 itself.

If Job ***CPU burst is > 24*** m sec then it is moved to last Queue. i.e. Q2 where Jobs are selected on FCFS basis Thus Jobs are selected from Q2 when Q0 and Q1 are empty. Once a Job is selected from Q2, it is executed completely till I/O burst is encountered.

Thus this algorithm gives highest priority for Jobs whose CPU burst time =8m sec .Jobs whose ***CPU burst time is> 8*** but =24 is given second highest priority. Jobs whose ***CPU burst time > 24*** m sec are given least priority.

1. **DEAD LOCKS**

Any process to perform some task it requires some resources. Therefore any process which requires resources it will request O.S fort that resources. If the resource is free then Operating System allocates the resource to the process. If it is not free then O.S will ask the process to wait until the resource is free by some other process. Thus any process during the execution will perform two operations. They are Resource Request and Resource release.

**Characterisation & Necessary Conditions for Deadlock**

**Deadlocks:**

Any process to perform some task it requires some resources. Therefore any process which requires a resource, it will request for that resource. If the resource is free then O.S. allocates the resource to the process. If, it is not free then Operating System will ask the process to wait until the resource is freed by some other process. Thus any process, during the execution, will perform two operations they are ***ResourceRequest*** and ***Resource Release****.*

**Definition of Deadlock:**

A set of process are said to be in dead lock state, when each process in the set is waiting for an event that can only we caused by another process in the set. The events are resource allocation and resource release. Thus in dead lock, process never finishes execution and resources are tied up, preventing other jobs waiting forever. Thus when a process is in dead lock state then we say that no work is done.

**Example 1:**

Let us assume a system contains 2 resources card reader and printer a process CARD READER and PRINTER. A process P1 holds card reader and request for printer. A process P2 holds printer and request for card reader. Now process P1 waits until printer is available and process P2 waits until card reader is available. Thus both processes wait forever. Now we say P1 and P2 are in deadlock.

**Example 2:**

There are three tape drives in an system. There are three processes; each holds one of these taped drives. Now if each process request for another tape drive then we get dead lock.

In dead lock the process will not do any work except sleeping. When a process requests a resources and if the resources are not available at that time the process enters into wait state. It may happen that waiting process will never change state because the resources they have required are held by other waiting process. This situation is called a deadlock

**Necessary Conditions for Dead Lock**

A dead lock can arise if and only if the following four conditions hold simultaneously in the system.

(1) Mutual exclusion

(2) Hold and wait

(3) Non-preemption

(4) Circular wait

(1) **Mutual exclusion:**

When one or more resources are held in non-sharable mode i.e. one process at a time can use the resource. If another process request for that resource, then the requesting process must be delayed until the resource is released. Thus dead lock occurs because the resources is non-sharable mutual exclusion condition must be imposed and hence we get deadlock .If the resource is sharable then mutual exclusion condition is need not be imposed. We will not get dead lock.

(2) **Hold & Wait:**

A process P1 holds resource R1 and request for resource R2. A process P2 holds resource R2 and request or resource R1. Then we say the process P1 and P2 are in deadlock.

In dead lock state, process sleeps i.e. wait. While waiting, process holds some resources these held resources are neither used by the process nor given to other processes. Thus the resources are kept idle, thus are to hold and wait, dead lock occur.

(3) **Non-preemption:**

Resources cannot be preempted, i.e. resources cannot be taken forcefully i.e. a resource can only be release voluntarily by the process and holding it after the process has completed its task. Thus due to non-preemption of resources dead lock occurs (If the resources are preempted dead lock will not occur).

(4) **Circular Wait**

There must exist a-set of waiting processors P0, P1.... Pn-1, Pn such that P0 holds a resource Ro and request for Rl, P1 holds R1and request for R2 so on Pn-1 holds Rn-1and request Rn and Pn holds Rn and request for Ro. Now we say P0, P1.... Pn-1, Pn are in dead lock.

If all the above four conditions are satisfied simultaneously then dead lock exists. If any one condition is not satisfied dead lock will not exists.

**Dead Lock Prevention**

To prevent dead lock, any one of the above four conditions should be denied. This is because, if at least one condition is not satisfied then deadlock will not occur.

(1) **Denying Mutual Exclusion**

Mutual exclusion occurs only for non sharable resource for example, printer is non-sharable resource. i.e. it cannot be simultaneously shared by more than one process. To prevent mutual exclusion, we make the resource sharable then dead lock never occurs as the resource can be shared by many processes.

Ex. Read only files are the best examples for sharable resource but if we make the resource sharable (like printer) then we get erroneous output. In general it is not possible to prevent dead lock by denying mutual exclusion condition. Since some resource should be compulsorily non-sharable.

**(2) Denying Hold & Wait**:

It is possible to prevent hold and wait. There are two techniques for this (1) every process should request all the resources that it requires for the entire execution before starting the execution only.

Ex. Suppose a process requires 4 resources for the entire execution then request all the resources before starting the execution. If all the four resources are free then start the execution by acquiring all the resources. If at least one resource is not free, do not start the execution.

This protocol has two disadvantages

1. Resource utilization poor since the resources which are required at the end of the execution are also acquired at the beginning only, these resources are kept idle for most part of the execution. i.e. these resources (i.e resources which are required at the end of the execution) are not used by the process for long time nor they are given to other processes.
2. Starvation is possible. A process that needs several resources may have to wait indefinitely even if at least one resource is allocated to some other process. Thus the process may not start execution at all.

To overcome these two disadvantages another protocol is developed.

Here any process can request for new resources only if it has none. For example, a process is having 4 resources. Now it requires two more resources then the process should release four resources before requesting for two new resources. If two resources are free, they are allocated to the process if they are not free process waits without holding any resources. Thus holding and wait is prevented.

**(3) Preventing non-preemption**

To prevent non preemption we use the following technique. A process P1 is having 4 resources. Now it request for two more resources. Now O.S. checks whether these two resources are free or not. If they are free, they are given to P1. If they are not free, then O.S. checks whether these two resources are tied up with which process. Let us assume they are tied up with P2 process. Now O.S. checks whether a process P2 is a sleeping or ready process. If it is sleeping process, then OS takes the two resources forcefully from process P2 and allocates them to P1. Now P1 uses all the six resources and complete its task. Suppose P2 is a ready process. Then OS cannot take the resource forcefully. In such a case P1 sleep. While P1 is sleeping other process can preempt the resources of P1 since P1 is sleeping.

**(4) Preventing Circular wait**

To prevent circular wait, we impose ordering of all resource types. i.e., we assign a unique integer number for each resource. This allows us to compare two resources and determine whether one precedes another in the ordering. The technique is each process can request for new resources in increasing order of assigned number. Initially, a process can request only no. of resources. Let us assume the process has acquired resource Ri after this, process can request for now resource Rj If and only of f(Rj)> f(Ri). Where f(Rj) and f(Ri) are the assigned numbers for the resources Ri and Rj. If f(Rj) <f(Ri) then the process must release Ri and request for Rj. After acquiring Rj , now the process can request for Ri. If this technique used circular wait cannot hold.

**Proof:**

Let us assume circular wait holds even if this technique is used.

Let us assume, there are four processes P0, P1. P2, P3. P0 holds R0 and request for R1 and P1 holds R1 and request for R2, P2 holds R2 and request for R3, P3 holds R3 and request for R4

The technique says that requesting resource number should be greater than the holding resource number.

f(R1) > f(R0)

f(R2) > f(R1)

f(R3)> f(R2)

f(R0)> f(R3)

Therefore f(R0) > f(R3)> f(R2) > f(R1) > f(R0)

* f(R0) > f(R0) which is false.

Therefore our assumption is wrong i.e. circular wait does not hold good.

**Dead Lock Avoidance**

When the technique dead lock prevention is used, it produces some side effects such as low device utilization (i.e. keeping the resources idle for long time) and reduce system throughput (i.e. a process is waiting for a resource, which is always allocated to some process or the other. This results in starvation). An alternative method for avoiding dead lock is using dead lock avoidance.

When a process request for some resources during execution, then we check whether these resources are available or not. If they are not available, OS asks the process to wait. If they are available, then system checks whether the allocation of these resources to the process results in safe state or not. A safe state is one where all the processes can complete the execution. An unsafe state is one, where some processes cannot complete execution at all. Safe state leads to no dead lock, an unsafe state may lead to dead lock.

When a process request for some resource, we check, whether these resources can be given to the process or not. Resources can be given to a process, only when the resulting state is safe. Otherwise we cannot give resources to the process, even though, they are free. In safe state, there will be some safe sequence (P2, P1, P3). This indicates that first P2 will complete execution and releases its resources, now P1 will take these resources completes execution and releases the resources. Now P3 will take these resources, complete execution and releases its resources. .. The resulting state is said to be safe, when the sequence of execution is (P2, P1, P3).

**Ex:** Let us assume there are 3 processes P1,P2,P3. Let us assume there are 3 resources all of same and maximum no. of resources are 12

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Max needed | Presently allocated | Available |
| P1 | 10 | 5 | 12-(5+2+2)=3 |
| P2 | 4 | 2 |
| P3 | 9 | 2 |

Let us verify whether the current state is safe or not. P1 and P3 cannot complete execution first. Since their need is> availability. But for P2 the need is < = availability. The first process that can complete execution is P2. After completion, P2 releases its resources.. Now availability is 5 (i.e. 3+2). Now P1 need is < = availability. Therefore P1 can complete execution. After completion, P1 releases its resource. Now availability is 10 (5+5). Now P3 need is < = availability P3 can complete execution and releases its resources. Thus all the processes are able to complete their execution. System is safe and safe sequence is (P2, P1, P3).

**Dead Lock Recovery**

When dead lock exists in the system, then the system must recover from deadlock. There are two methods for breaking dead lock.

(1) Process termination (2) Resource preemption

**(1) Process termination**

**(a) Kill all dead lock processes**

Here all the processes which are dead locks are killed and the resources they are holding are released. The advantage of this method is, it is easy to implement. But the disadvantage is that we are losing processes which are computed for long time

**(b) Kill one process at a time until dead lock cycle is eliminated.**

Here only one process at a time is killed and the resources held by this process are released. Now call deadlock detection algorithm and check whether deadlock is present or not. If it is still present, kill one more process and again call deadlock detection algorithm. Repeat this method until there exists no dead lock.

To kill a process among dead lock processes. We kill that process which is executed for small amount of time. We should never kill a process which is executed for large amount of time as most of the execution of the process is completed. After killing the process, the process goes to end of ready queue for starting the execution again.

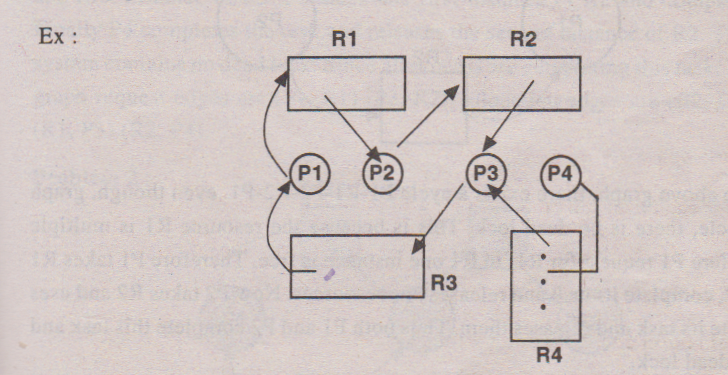
**(2) Resource Preemption:**

Here we successively preempt some resources from the processes and give these resources to another processes until deadlock cycle is broken. The question is which resource must be preempted? The selection criteria depends on priority of processed no. of resources, a process still requires to complete its task and how many processes are requesting for that resource. Usually we kill that process which is having large no. of resources. So that all these resources are released and given to other processes. Now these other processes can complete execution and thereby dea4 lock cycle is broken.

**Resource Allocation Graph**

To detect whether deadlock occurs in the system or not we use resource allocation graph. This graph indicates which process is requesting for which resource and which resource is allocated to which process. Resource allocation graph is directed graph G G=(V,E) where Vi is a vertex set . E is the Edge set. Vertex set is partitioned into 2 sets P and R. P={p1,p2…..pn} where p1,p2……pn are processes where r1,r2….rn are resources.

A process is indicated by circle. Resource is indicated by square. The square contains one or more dots (.), indicating resource of some type. E is the edge set, which contains two types of edges (Pi, Ri) and (Ri, Pi). (Pi, Ri) is called request edge, indicating process Pi is requesting for resource Ri (Ri, Pi) is allocation edge, indicating resource Ri is assign to process P1



Here the request edges are (P1, R1) (P2, R2) (P3, R3) allocation edges are (R1, P2) (R2, P3) (R3, P1) (R4, P4).

If a graph contains cycle and cycle includes resources of single instance (i.e. one dot) then cycle in the graph implies dead lock.

In the above graph, there is a cycle.

P1 🡪 R1🡪P2 🡪 R2🡪 P3🡪 R3🡪 P1

In this cycle the resources included are R1, R2, R3. They are all single instance. Therefore system contains dead lock.

Suppose the graph contains cycle and the resources in the cycle are multiple instances, then cycle in the graph does not necessarily imply the dead lock has occurred.

**Dead Lock Detection**

Suppose dead lock prevention and dead lock avoidance are not used then there is a possibility of dead lock occurring in the system. Hence one should detect whether dead lock present in the system or not at regular intervals time. If it is present then OS should call dead lock recovery.

**BANKERS ALGORITHM**

When a process request for some resources, which are currently available then the system decides whether the allocation of these resources will leave the system is safe state or not. If the system is in safe state then the request made by the process for resources can be granted. Otherwise, the request cannot be granted.

Bankers algorithm will not check whether the system is in safe state or not its job is just allocate resources to the processes if they are available. After allocating the resources one Bankers algorithm will call safety algorithm.

**Data Structures Used are as follows**

Let there be n processes, and m resources.

**Max matrix**: it is **n x m** matrix defining the maximum demand of each process. If max (i, j) = k ; then it means process Pi is requesting for K instances of resource rj

**Allocation matrix**

It is n x m matrix defining the no. of resources of each type currently allocated to each process. If allocation (i, j) = K; then it means K instances of resource rJ i already allocated to process Pi.

**Need Matrix**

It is **n x m** matrix indicating the no. of resources required by each process. If need (i, j) = K means process Pi still requires K instances of resource rj. Note that ***need (i, j) = max (i, j) - allocation (i, j).***

**Available**

It is a vector of length m indicating the no. of available resources of each type. If available (j) = K then it means, there are K instances of resource rj free.

**Request**

It is a vector of length m of process C. if request (J) = K then it means process Pi wants K instances of resource rj. When a request for resources is made by process Pi the following actions are taken.

**Algorithm**

1. If request i <= need i t

then go to step 2

else there is an error

stop

1. If request i < = availability

then go to step 3

else make the process Pi wait

1. allocation i =allocation i + request i

need i = need i - request i

available = available - request i

**SAFETY ALGORITHM**

This algorithm checks whether the current state is safe or not.

1. Max

2. allocation

3. Need

4. Available

5. **Work** is a vector of length m (m for no. of resources) and initial value of work is available.

Whatever modification are to be perform out on available, perform on work i.e. don’t modify

available

6. **Finish** is a vector of length n(n is no. of process) initially finish contains (fff ….n times). False indicates process have not complete the execution true indicates processes have. Completed the execution. Thus as and when process completes execution, changes the corresponding entry in Finish vector to true.

**Algorithm**

* 1. Work = available

Finish (i) <- false

for i=1,2,…n

* 1. If need ≤ work & finish[i]< -false

then go to step 3

else go to step 4

* 1. Work=work +allocation

Finish[i]<- true

go to step2

* 1. If finish[i]<- true for all

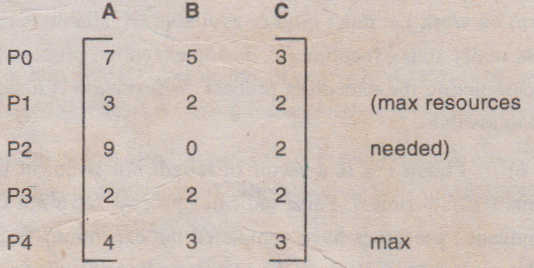
then system is safe

else system is not safe

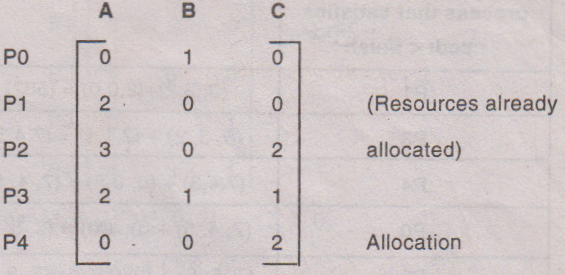
**Deadlock problem**

**Problem:**

Consider a system with 5 processes P0, P1, P2, P3, P4 and there are 3 resources A, B, C resource A has 10 instances, resource. B has 5 instances, resource C has 7 instances. Let the max matrix be

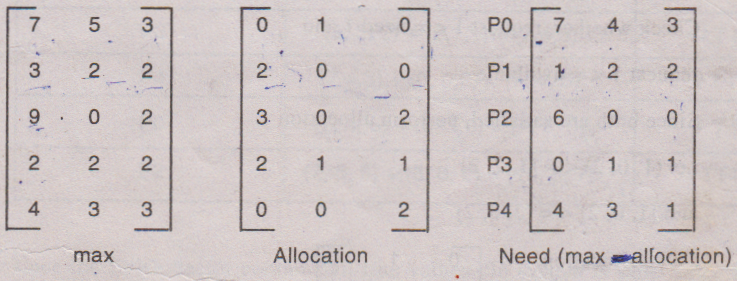


Let the allocation matrix be



1. Check whether the present state is safe or not. If so give safe sequence (safety algorithm).
2. If P1 request (1, 0, 2) can it be granted? If so what is the safe sequence.

**Solution:**



Available = [10- (2+3+2), (5- (1 + 1)), (7- (2+ 1+2))]

= [3, 3, 2]

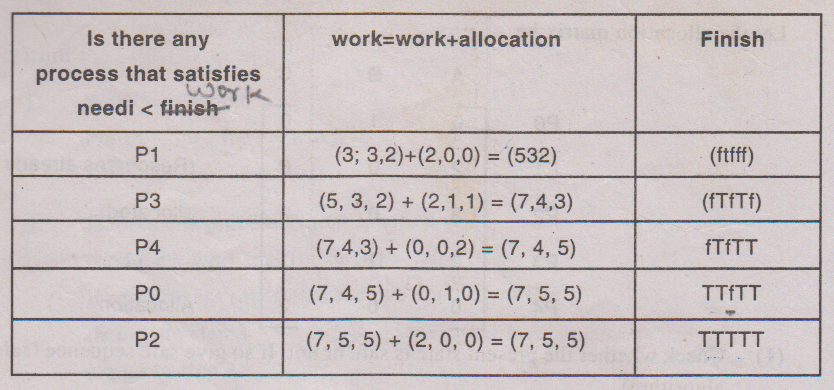
Initially work = available = [3, 3, 2]

i.e. 3 resources of type A are free

3 resources of type B are free

2 resources of type C are free

Finish = (f f f f f)



Since the finish vector contains all trues system is safe.

Safe sequence is (P1, P3, P4, P0, P2)

1. Here P1 is requesting for (1, 0, 2)

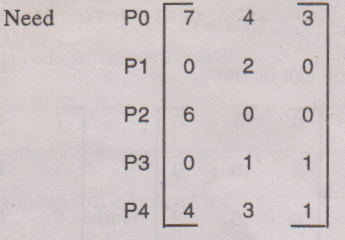
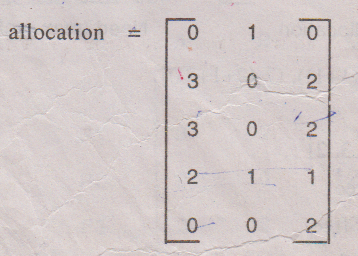
1 request 1 (P1) = (1, 0, 2)

Check whether, request 1 <= need 1 and

request 1 <= available

Since both are satisfied, perform allocation

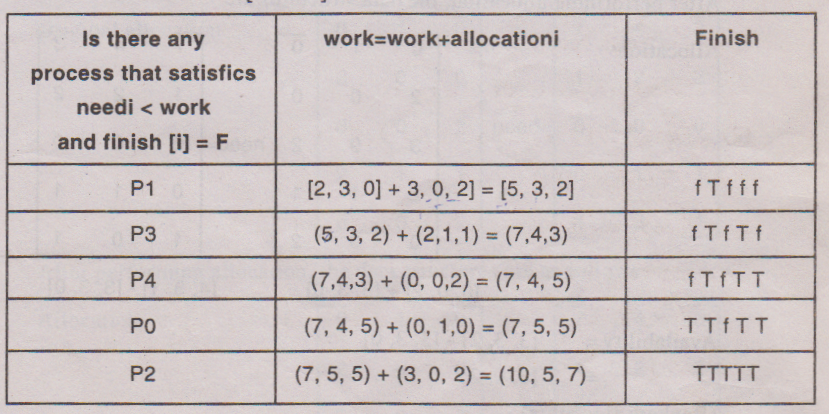
i.e. (1, 0, 2) <= [1, 2, 2] and [1,0, 2] <= [3, 3,2]



available = [3,3,2]-[ l, 0, 2] = [2,3,0]

work = available =[2, 3, 0]

Finish= [f f f f f]



Since the finish vector contains all true values. So system is safe. Since system is safe the request can be granted.

Safe sequence is (P1, P3, P4, P0, P2)