Process :

When a program resides on disk then it is simply a program and when it is loaded into main memory for execution, it becomes a process and then it competes for resources such as CPU, memory, files etc. Process is a program in execution.

The OS allocates system resources and keep track of all active processes. In multi-processing system, a no. of active processes are in various stages of execution at any point. The OS keeps track of the state of each process and records all the state changes as they occur.

The process management provides the following functions :

1) creating and removing process.
2) controlling the progress of a processes.
3) tracing the interrupt during the programs execution
4) Allocating hardware resources among processes
5) Providing synchronization and communication signals among processes

Process States :

As a process executes, it changes state. The state of a process is defined

by current activity of that process.

New :- The process is being created.
Running :- Instructions are being executed.
Waiting :- The process is waiting for some event to occur (such as an I/O completion)
Ready :- The process is waiting to be assigned to a processor.
Terminated :- The process has finished execution.

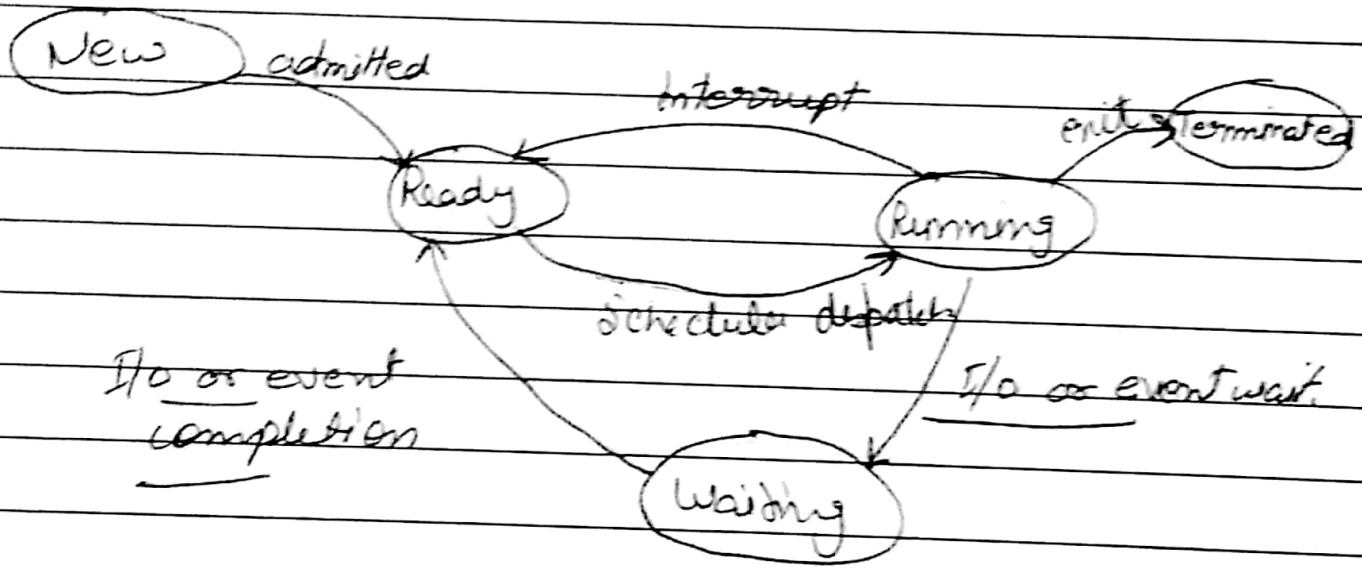
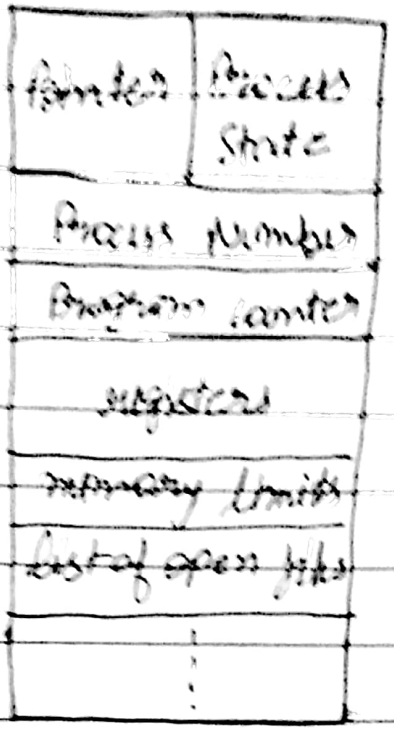


Diagram of process state.

Process Control Block (PCB)

Each process is represented in the operating systems by a process control block (PCB) also called task control block. It contains many pieces of information

associated with a specific process.



PCB

⇒ Process States: The state may be new, ready, running, waiting ... etc.

⇒ Program counter:— The counter indicates the address of the next instructions to be executed for this process.

⇒ CPU registers:- The registers vary in numbers and type, depending on the computer architecture. They include accumulators, index register, stack pointers and general purpose registers.

Dt...../...../....

⇒ CPU-Scheduling Information:- This information includes a process priority, pointers to scheduling queues and other scheduling parameters.

⇒ Memory-management information:- This information may include such information as the value of the base and limit registers, the page tables depending on the memory system used by the OS.

⇒ Accounting information:- This information includes the amount of CPU, time limits, account numbers, job or process numbers, and so on.

⇒ I/O Status information:- The information includes the list of I/O devices allocated to the process a list of open files etc.

/

CPU Scheduling

In multiprogramming A process is executed until it must wait for completion of I/o request In simple computer CPU sit idle; all this waiting time is wasted. With multiprogramming, when one process has to wait, the os takes the CPU away from that process and give the CPU to another process. This pattern continues.

Scheduling is the fonction of OS. Almost all computer resources are scheduled before use.

CPU – I/O Burst Cycle's

Process execution consists of a cycle of CPU execution and I/O wait. Process alternate b/w these two states. Process execution begins with a CPU burst. That is followed by I/O burst, then another CPU burst, then another I/O burst, and so on. The CPU burst will end with a system request to terminate execution, rather than another I/O burst.

Dt...../...../.....

load store
add store. CPU burst } CPU burst
read from file

wait for I/O } I/O burst

store increment
index
write to file } CPU burst

wait for I/O } I/O burst

load store
add store
read from file } CPU burst

wait for I/O } I/O burst.

Alternate Sequence of CPU & I/O bursts.

CPU Scheduler:

When the CPU become idle, the OS must select one of the processes in the ready queue to be executed. The selection of process is carried out by short-term scheduler or CPU scheduler. The CPU scheduler selects from among the processes in memory that are ready to execute and allocates the CPU to one of them.

Preemptive Scheduling :-

Non-preemptive scheduling :-

Once the CPU has been allocated to a process, the process keeps the CPU until its termination or its goes to the waiting states.

Preemptive Scheduling :-

Here even if the CPU has been allocated to a certain process, it can be preempted from this process any time either due to time constraint or due to priority reasons.

Content Switching :-

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a content switch. The content of a process is represented in the PCB of a process; it includes the value of CPU registers, the process states and memory management information. When a content switch occurs, the Kernel saves the content of old process in its PCB and loads the saved content of the new process scheduled to run.

Scheduling Criteria's

① CPU utilization: we want to keep the CPU as busy as possible. The CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40% to 90%

② Throughput:- No. of processes completed per time unit called throughput.

~~③ The to~~

③ Turnaround time:- how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Turn around time = time of completion of job − time of submission

④ Waiting time:- CPU-scheduling algo. does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue. It is the sum of the time intervals for which the process has to wait in the ready queue.

⑤ Response time:- is the amount of time it takes to start responding, but not the time that it takes to output that response.

We want maximize CPU utilization and throughput and to minimum turnaround time, waiting time, and response time.

Scheduling Algorithms:-

① First come first-served (FCFS):-

With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS is managed with a FIFO queue. When the process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue.

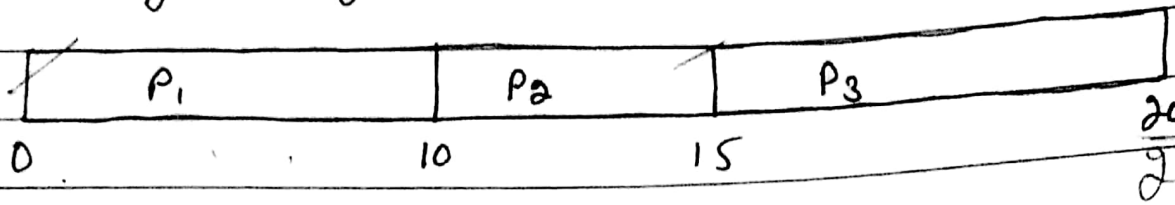
The average time under FCFS is quite long.

E.g.:-

Process	Burst time (in ms)	AT
P_1	10	0
P_2	5	1
P_3	5	2

If the process arrived in order P_1, P_2, and P_3, then the avg. waiting time for the process

avg waiting time:



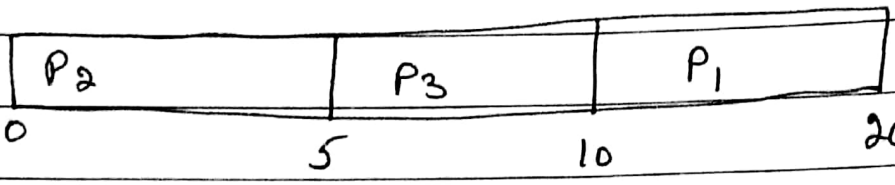
waiting time for $P_1 = 0$

" $P_2 = 10$

$P_3 = 15$

Avg. waiting time $= \frac{0+10+15}{3} = \frac{25}{3} = 8.33$ m.s.

But if process arrive in order P_2, P_3 and P_1



Avg. waiting time $= \frac{0+5+10}{3} = \frac{15}{3} = 5$ ms.

Avg waiting time will always depend upon the order in which the processes arrive. This algo. never recommended where performance is a major issue.

The FCFS is non-preemptive. Once ~~Shortest Job First Scheduling:~~ 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.

DI...../...../.....

② Shortest-Job-First Algorithm:-

This algorithm associates with each process the length of the next CPU burst. When CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

Advantage:- It gives the minimum average waiting time.

Disadvantage:-

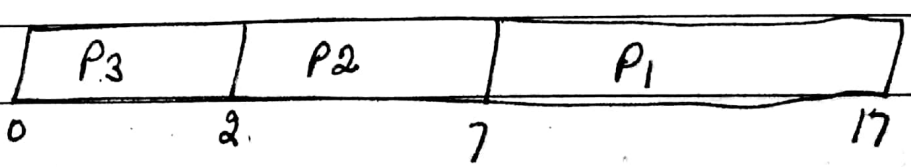
The problem is to know the length of time for which CPU is needed by a process. A prediction formula can be used to predict the amount of time for which CPU may be required by a process.

e.g.s

P	BT
P1	6
P2	8
P3	7
P4	3

Process	CPU-burst	
P_1	10	0
P_2	5	1
P_3	2	2

With non-preemptive SJF

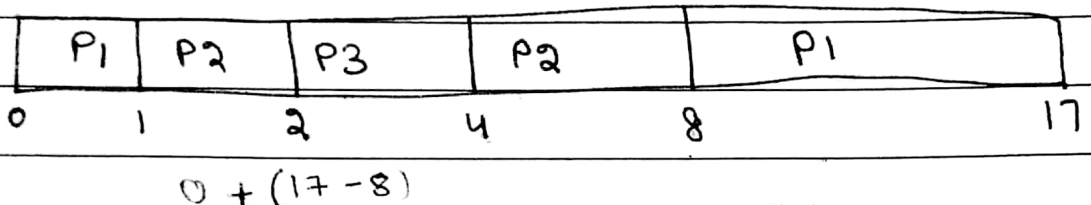


(Gantt Chart)

Avg. waiting time = $\frac{0+2+7}{3} = \frac{9}{3} = 3$ milliseconds

If the process arrived in the order P_1, P_2, P_3 and at the time as mentioned, then avg. waiting time using preemptive SJF

		Time of Arrival.
P_1	10-9	0
P_2	15-14	1
P_3	2	2



0 + (17 - 8)

AT - CT

Waiting time for P_1 = 0 + (8 - 1) = 7

" " P_2 = 1 + (4 - 2) = 3

" " P_3 = 2

Avg. waiting time = $\frac{7+3+2}{3} = \frac{12}{3} = 4$ ms.

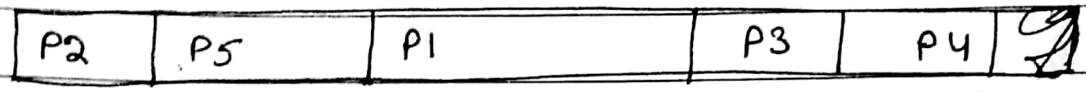
(3) Priority Scheduling :-

A priority is associated with each process and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS order.

Turnaround Time = Completion time - Arrival time

Waiting Time = Turnaround time - Burst time

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



Avg waiting time is 8.2 ms.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority-scheduling algo. will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling algo. will simply put the new process at the head of the ready queue.

A major problem with priority scheduling is indefinite blocking or starvation. A priority-scheduling algo. can leave some low-priority processes waiting indefinitely for the CPU.

In this problem higher priority processes can prevent a low-priority process from ever getting the CPU.

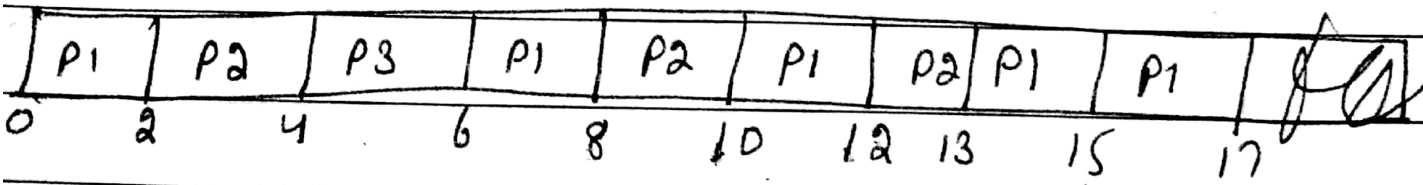
A solution to the problem of indefinite blockage of low-priority processes is aging. Aging is the technique of increasing the priority of processes that wait the system for a long time.

Round-Robin Scheduling:-

RR scheduling algorithm is designed especially for time sharing systems. It is similar to FCFS scheduling, but preemption is added to switch b/w processes. A small unit of time, called a time quantum or time slice. is defined. Its implementation requires timer interrupts based on which the preemption takes place.

e.g:-

Process	CPU-Burst time
P1	10
P2	5
P3	2



Time quantum is 2 milli-seconds

RR Algo.

waiting time for P1 will be = 0 + (6-2) + (10-4) + (13-12)
= 4 + 2 + 1 = 7 ms

" " P2 = 2 + (8-4) + (12-10)
= 2 + 4 + 2 = 8 ms.

" " P3 = 4

Avg. waiting time = $\frac{7+8+4}{3} = \frac{19}{3} = 6.33$ ms.

The performance of this algo. depends upon many factors:

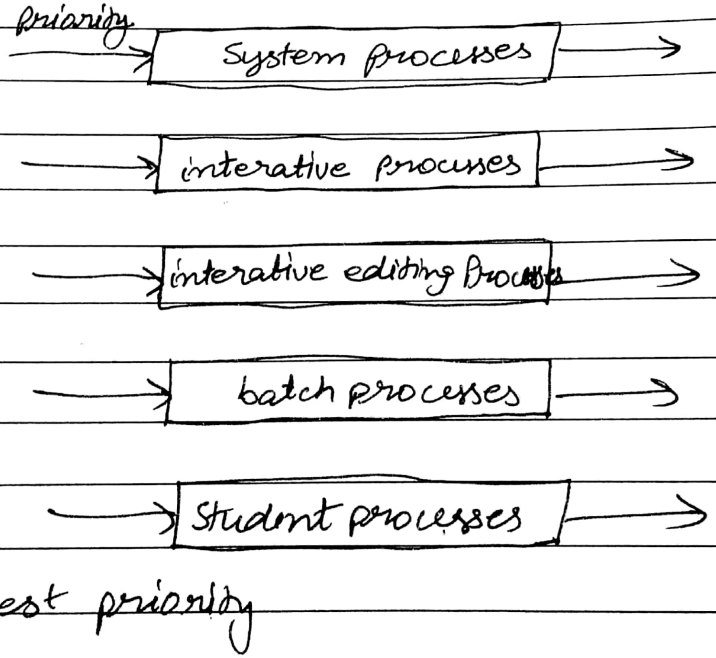
① Size of time quantum.
* If time slice size is large then algo. becomes same as FCFS algo and this performance degrades.
* If time quantum's size is very small then the no. of context switchs increases which slow down the overall execution of all the processor which is not acceptable.
So. the time quantum should not be very large and also should not be too small to achive good system performance.

② The no. of context switches should not be too many to slow down the overall execution of all the processes.

Multilevel Queue Scheduling :-

Multilevel queue scheduling has been created for situations in which processes are classified into different groups.

Multilevel queue-scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of processes such as memory size, process priority or process type. Each queue has its own scheduling algorithm. There must be scheduling among the queues.



Each queue has priority over lower-priority queues. No process in the batch queue could run unless the queues for system processes, interactive processes and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running the batch process would be preempted.

Multilevel Feedback Queue Scheduling:-

In multilevel queue scheduling algorithm, processes are permanently assigned to a queue. Processes do not move b/w queues. This setup has the advantage of low scheduling overhead but the disadvantage of inflexible.

Multilevel feedback queue scheduling allows a process to move b/w queues. The idea is to separate processes with different CPU-burst. If a process uses too much CPU time, it will be moved to a lower-priority queues. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. Ildy, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

Dt...../...../.....

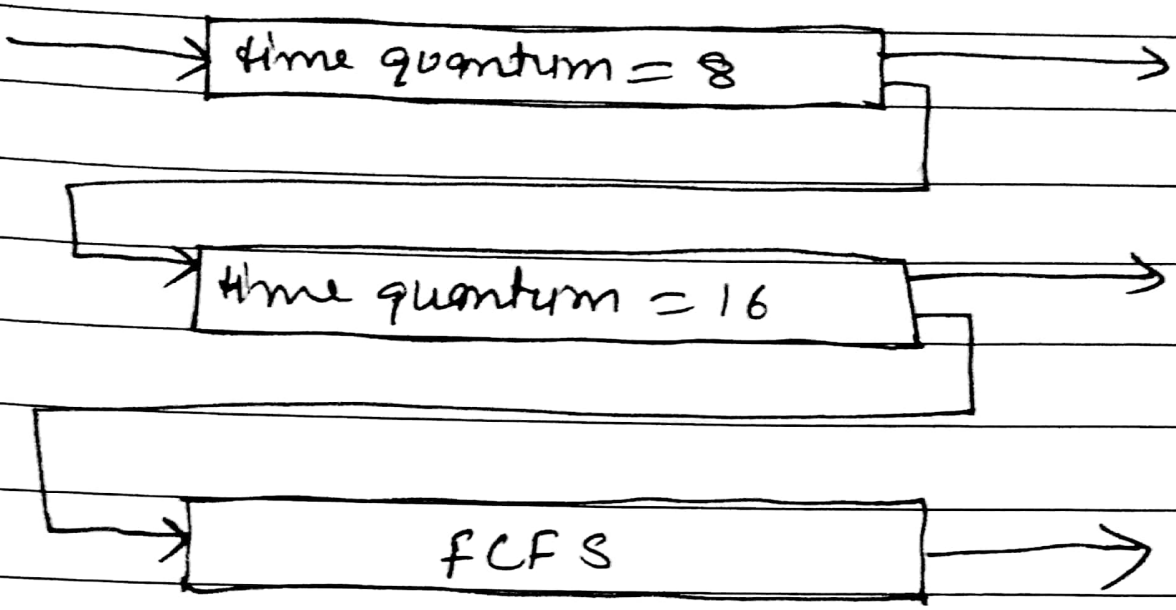
e.g:-

Consider a multilevel feedback queue with three queues, numbered from 0 to 2. The scheduler first executes all processes in queue 0. only when queue 0 is empty will it execute processes in queue 1. Why, processes in queue 2 will be executed only if queues 0 and 1 are empty.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. The queue 1 is given a time quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis, only when queues 0 and 1 are empty.

⇒ Multilevel feedback queue is defined by the following parameters:-

* The number of queues.
* The scheduling algorithm for each queue.
* The method used to determine when to upgrade a process to a higher priority queue.
* The method used to determine when to demote a process to a lower-priority queue.
* The method used to determine which queue a process will enter when that process needs servi



Multilevel feedback queues.

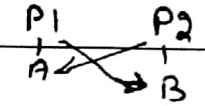
Dt...../...../.....

Deadlock

A process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

eg:-

Process1 has been allocated non-sharable resources A (tape drive) and P2 has been allocated non-sharable resources B (Printer). Now if P1 needs resource B (Printer) to proceed and P2 needs resources A (tape drive) to proceed and there are only two processes in the system, each blocked the other and all useful works in the system stops. This situation is termed deadlock.



Deadlock Characterization

(for deadlock to occur)

Necessary Conditions:

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

Mutual Exclusion: At least one resource must be held in a nonsharable mode i.e. only one process at a time can use the resource. (only one at a time)

If another process request 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 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 exist such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2 ... P_{n-1} is waiting for a resource that is held by P_n and P_n is waiting for a resource that is held by P_0.

Definition:

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

Note: If a system has two CPU, then the resource type CPU has two instances. Ily resource type printer may have five instances.

* Request:
* Use:
* Release:

Resource-Allocation Graph:

Deadlock can also be described in terms of directed graph called a system resource-allocation graph. This graph consist of a set of vertices V and set of edges E. The set of vertices V is partitioned into two different types of nodes $P = \{P_1, P_2 \dots P_n\}$, the set consisting of all the active processes in

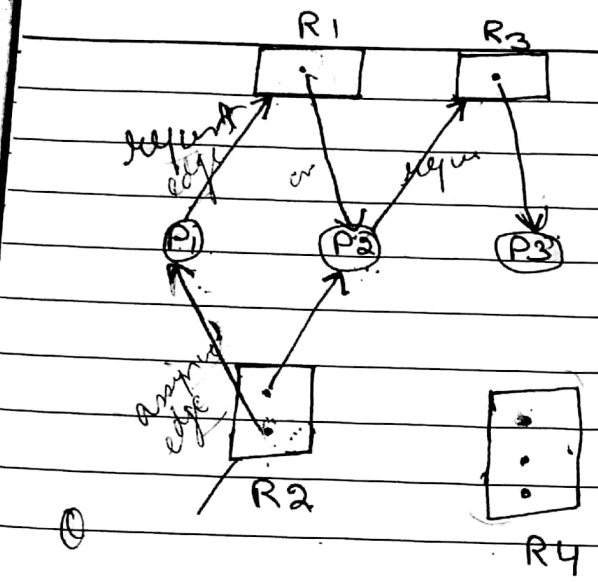
the system and $R = \{R_1, R_2 \ldots R_m\}$, the set consisting of all resource types in the system.

* A directed edge from process P_i to resource type R_j ($P_i \rightarrow R_j$), it signifies that process P_i is requested an instance of resource type R_j and is waiting for that resource.

* A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i.
* A edge $P_i \rightarrow R_j$ is called request edge.
* A directed edge $R_j \rightarrow P_i$ is called assignment edge.
* We represent each process P_i as a circle and each resource type R_j as a square.
* Resource type R_j may have more than one instance and such instances are represented by dot within the square.
* When Process P_i request an instance of resource type R_j a request edge is inserted in the graph and when this request is fulfilled, the request edge is transformed to an assignment edge.

$P \rightarrow R$ — Requ

$P \leftarrow R$ — assn.



* The sets P, R and E

P = {P1, P2, P3}

R = {R1, R2, R3, R4}

E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

* Resource instances:

⇒ 1 instance of resource type R1

⇒ 2 --- R2

⇒ 1 --- R3

⇒ 4 --- R4

Resource allocation graph.

Process states:

* Process P1 is holding an instance of resource type R2 and is waiting for R1.
* P2 is holding an instance of R1 and R2 and is waiting for R3.
* P3 is holding an instance of R3.

* If the graph contains no cycles, then no process in the system is deadlocked. If the graph contain a cycle, then 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

Dt...../...../.....

necessary and sufficient condition for the existence of a deadlock.

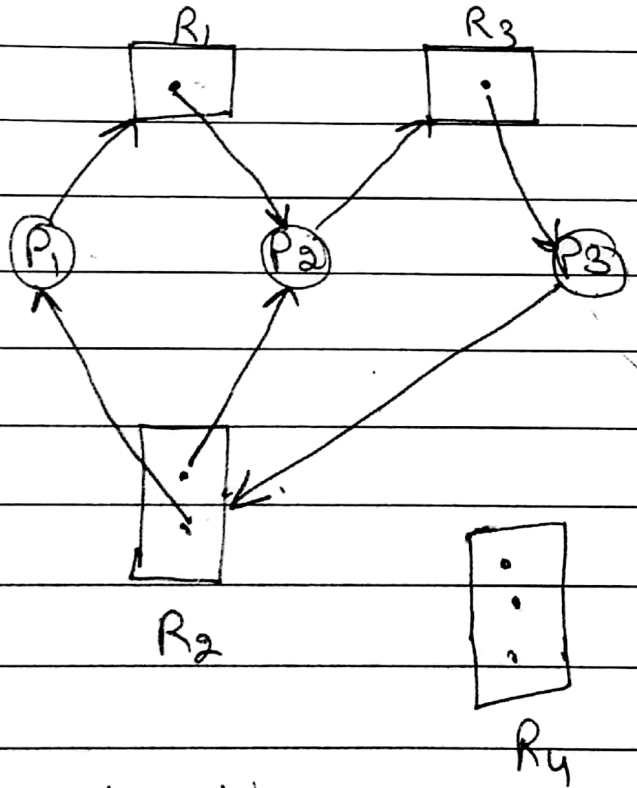
If the resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

eg:-

Let P3 request an instance of Resource type R_2. Since no resource instance is currently available.

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

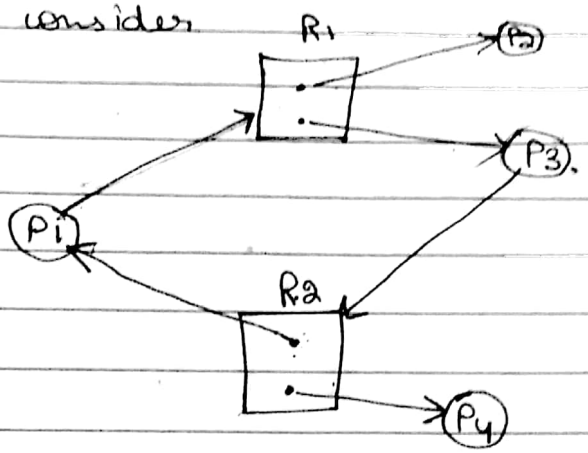
$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$



Resource - Allocation graph - with deadlock

Processes P1, P2 and P3 are deadlocked. P2 is waiting for R3 which is held by P3. P3 is waiting for either P1 or P2 to release resource R2. P1 is waiting for P2 to release resource R1.

Now consider



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

There is no deadlock. Process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

Definition of Deadlock

Resources are being held by processes that cannot run and more and more processes as they make requests for resources enter a deadlock state. The system will stop functioning and will need to be restart.

Methods for handling deadlocks:-

1. Deadlock Prevention :- is a set of methods for ensuring that at least one of the necessary conditions cannot hold.

2. Deadlock Avoidance :- requires that the OS be given in advance information concerning which resources a process will request and use during its lifetime. With this knowledge we can decide whether or not process should wait. To decide whether the current request can be satisfied or ~~not~~ delayed, the system consider the resources currently available, the resources currently allocated to each process and the future requests and release of each process.

3. Deadlock detection and Recovery :- Detect deadlock when it occurs and take steps to recover.

Dt...../...../.....

Deadlock Prevention :-

By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

① Mutual Exclusion : (Hold)

* This condition must hold for nonsharable resources. e.g printer which cannot be ~~shared~~ simultaneously shared by several processes.
* While nonsharable resources do not require mutual exclusion access and cannot be involved in a deadlock. Read-only files are example of a sharable resource. A process never needs to wait for a sharable resource.

② Hold and wait : (Not hold)

To ensure that hold and wait condition never occurs in the system. 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 the process has none. A process may request for resource and

use them Before it can request any additional resources it must release all the resources that it is currently allocated.

Ex:- A process that copies data from tape drive to disk file, sorts the disk file and then prints the result. If all resources must be requested at the beginning of the process, then process initially request the tape drive, disk file and printer, even though printer need only at the end.

* The second method allows the process to request initially only the tape drive and disk file. It copies from tape drive to disk file and then release both the resources. The process again request the disk file and printer. After printing it releases both the resources.

③ No preemption :- (Not Hold)

In this there is no preemption of resources that have already been allocated. To ensure that this condition does not hold we use the following protocol.

* If a process is holding some resources and request another resource that cannot be allocated to it, then all resources currently being held are preempted. The preempted resources are added to the list of resources for which process is waiting. This protocol is often applied to resources

whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot be applied to resources as printers and tape drives.

(4) Circular wait :- Circular wait One way to ensure that this condition never holds is to impose a total ordering of all resources types, and to requires that each process requests resources in an increasing order

(not hold)

Let $R = \{R_1, R_2 \dots R_m\}$ be the set of resource types. We assign each resource type unique number, which allows us to compare ~~to compare two resources~~ two resources and to determine whether one precedes another in our ordering.

$F(\text{tape drive}) = 1,$

$F(\text{disk drive}) = 5,$

$F(\text{printer}) = 12.$

P1

Each process can request resources only in increasing order. A process can initially request any number of instances of resource type R_i. After that the process can request instances of resource type R_j if and only if $R_j > R_i$

A process that wants to use the tape drive and printer at the same time must first request

the tape drive and then request the printer. /

Deadlock Avoidance:-

The side effect of prevention algo. is low device utilization and reduced system throughput. Deadlock-prevention algo., prevent deadlocks by restraining how requests can be made.

* A deadlock-avoidance algo. examines the resource-allocation state to ensure that a circular-wait condition can never exist.

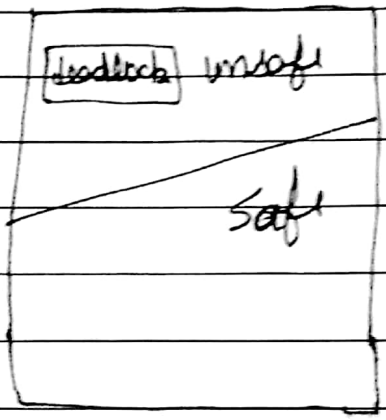
* The resource-allocation state is defined by the no. of available and allocated resources and the maximum demands of the processes.

Safe State:

A state is safe if the system can allocate resources to each process in some order and avoid a deadlock. A system is in a safe state only if there exists a safe sequence. A sequence of processes $\langle P_1, P_2, \ldots P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i, the resources that P_i can request can be satisfied by the currently available resources plus the resources held by all the P_j with $j < i$. If the resources that process P_i needs not available, then P_i can wait until P_j have finished. when they

have finished, P_i obtain all of its needed resources, complete its tasks, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources and so on ——

* If no such sequence exists, then the system state is said to be unsafe.



* A safe state is not a deadlock state.
* A deadlock state is an unsafe.
* An unsafe state may lead to a deadlock.

Ex:-

A System with 12 magnetic tape drives & 3 processes: P_0, P_1, P_2

	Max needs	current needs
P_0	10	5
P_1	4	2
P_2	9	2

12
3
9
5

$\langle P_0, P_2, P_0 \rangle$

P_1

At time t_0

P_0 is holding 5 tape drives

P_1 " 2 "

P_2 " 2 "

There are 3 tape drives free.

At time t_0 system is in a safe state.

sequence $\langle P_1, P_0, P_2 \rangle$ satisfies the safety condition.

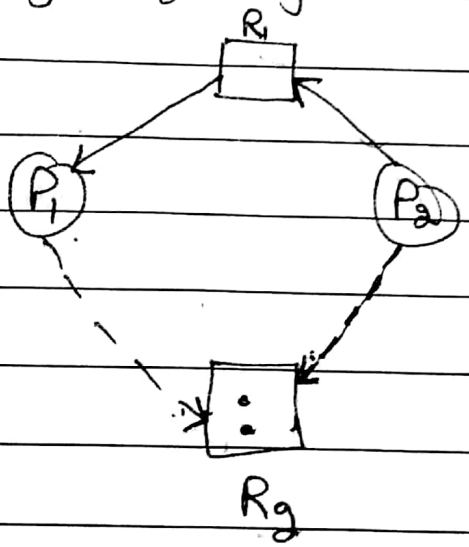
* P_1 can immediately be allocated all its tape drive & then returns them. total $3+2=5$ remaining
* P_0 get drive and return them. total $5+5=10$ rem
* P_2 get all drives & return them. total $10+2=12$ remaining

* A system may go from safe state to unsafe state.
* Suppose at time t_1, P_2 request and allocate 1 more tape drive. The system is no longer in a safe state.
* At this point only process P_1 can be allocated all its tape drives. P_0 is allocated 5 drives & has max of 10, it may request 5 more drives. & they are unavailable, P_0 must wait.
* Why P_2 may request an additional 6 drives and have to wait, resulting in a deadlock.

Resource-Allocation Graph Algorithm:

In addition to the request edge and assignment edges, we introduce a new type of edge, called a claim edge.

* A claim edge $P_i \rightarrow R_j$ indicates that P_i may request resource R_j at some time in the future.
* This edge resembles a request edge in direction, but is represented by a dashed line.
* When process P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to a request edge.
* Why when a resource R_j is released by P_i, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$



Other contents are same as the resource-allocation graph which is explained in previous topic

Banker's Algorithm :-

* The resource-allocation graph algo is not applicable to a resource-allocation system with multiple instances of each resource type. The banker algorithm is applicable for multiple instances but is less efficient.

This algo is used in banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

* When a new process enter 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 resources.

* Several data structures be maintained to implement the banker's algorithm. Let n be the no. of processes in the system and m be the no. of resources types. We need the following data structures:

* Available :-

A vector of length m indicates the no. of available resources of each type. If Available[j] = 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] = 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] = 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] = k, then process Pi may need k more instances of resource type Rj to complete its task.

Need[i,j] = Max[i,j] - Allocation[i,j]

P0 P2 P6 P0 P3

Safety Algorithm:

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length m and n. Initialize Work = Available and Finish[i] = false for i = 1, 2, ---n.

2. Find an i such that both
   a. Finish[i] = false
   b. Need$_i$ ≤ Work = Available.
   If no such i exist, go to step 4.
3. Work = Work + Allocation$_i$
   Finish[i] = true
   goto step 2.
4. If Finish[i] = true for all i, then the system is in safe state.

Resource-Request Algorithm:

Let Request$_i$ be the request for Process P_i. If Request$_i$[j] = k, then process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

Dt...../...../.....

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

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

If the resulting resource-allocation state is safe, the transaction is completed and process P_i is allocated its resources.

Ex. 5 A system with five processes P_0 thru. P_4 and three resource types A, B, C. Resource type A has 10 instances, Resource type B has 5 instances and resource type C has 7 instances.

At time T_0

	Allocation	Max	Available	Instances
	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	3 3 2 ✓	A = 10
P_1	2 0 0	3 2 2		B = 5
P_2	3 0 2	9 0 2		C = 7
P_3	2 1 1	2 2 2		
P_4	0 0 2	4 3 3		

What will be the content of need matrix.

(i) Need is defined to be Max − Allocation

	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

(ii) Is the system in safe state? If yes, what is the safe sequence.

* Applying safety algorithm on the given system.

for P_i, if $need_i \leq Available$

then P_i is in safe sequence.

$Available = Available + Allocation_i$

For P_0, $Need_0 = 7, 4, 3$

$(i = 0)$

$Available = 3, 3, 2$ $Need_i \leq Available$

condition is false

So P_0 must wait.

For P_1,

($i=1$) $Need_1 = 1, 2, 2$

$Available = 3, 3, 2$

$Need_1 < Available$ (True)

So P_1 will be kept in safe sequence.

Now Available will be updated as:

$Available = 3, 3, 2 + 2, 0, 0 = 5, 3, 2$

For P_2

($i=2$) $Need_2 = 6, 0, 0$

$Available = 5, 3, 2$

Condition is again false so P_2 must wait

For P_3

($i=3$) $Need_3 = 0, 1, 1$

$Available = 5, 3, 2$

Condition is true; P_3 will be in safe sequence

$Available = 5, 3, 2 + 2, 1, 1 = 7, 4, 3$

For P_4

($i=4$) $Need_4 = 4, 3, 1$

$Available = 7, 4, 3$

$\therefore$ $Need_i < Available$ true

So P_4 is in safe Sequence and

$available = 7, 4, 3 + 0, 0, 2 = \underline{7, 4, 5}$.

Now, we have two process $\underline{P_0}$ and $\underline{P_2}$ in waiting state

P_2 whose $Need = 6, 0, 0$

$Available = 7, 4, 5$

$Need_2 < Available$

P_2 comes in safe sequence

Available = 7, 4, 5 + 3, 0, 2 = 10, 4, 7

Next P_0 whose Need = 7, 4, 3

Available = 10, 4, 7

∵ $Need_0$ < Available

So, P_0 now comes in safe sequence

Available = 10, 4, 7 + 0, 1, 0 = 10, 5, 7

So safe sequence is < P_1, P_3, P_4, P_2, P_0 >

and system is in safe sequence.

Deadlock Detection :

A single instance of each resource type as well as systems with several instances of each resource type.

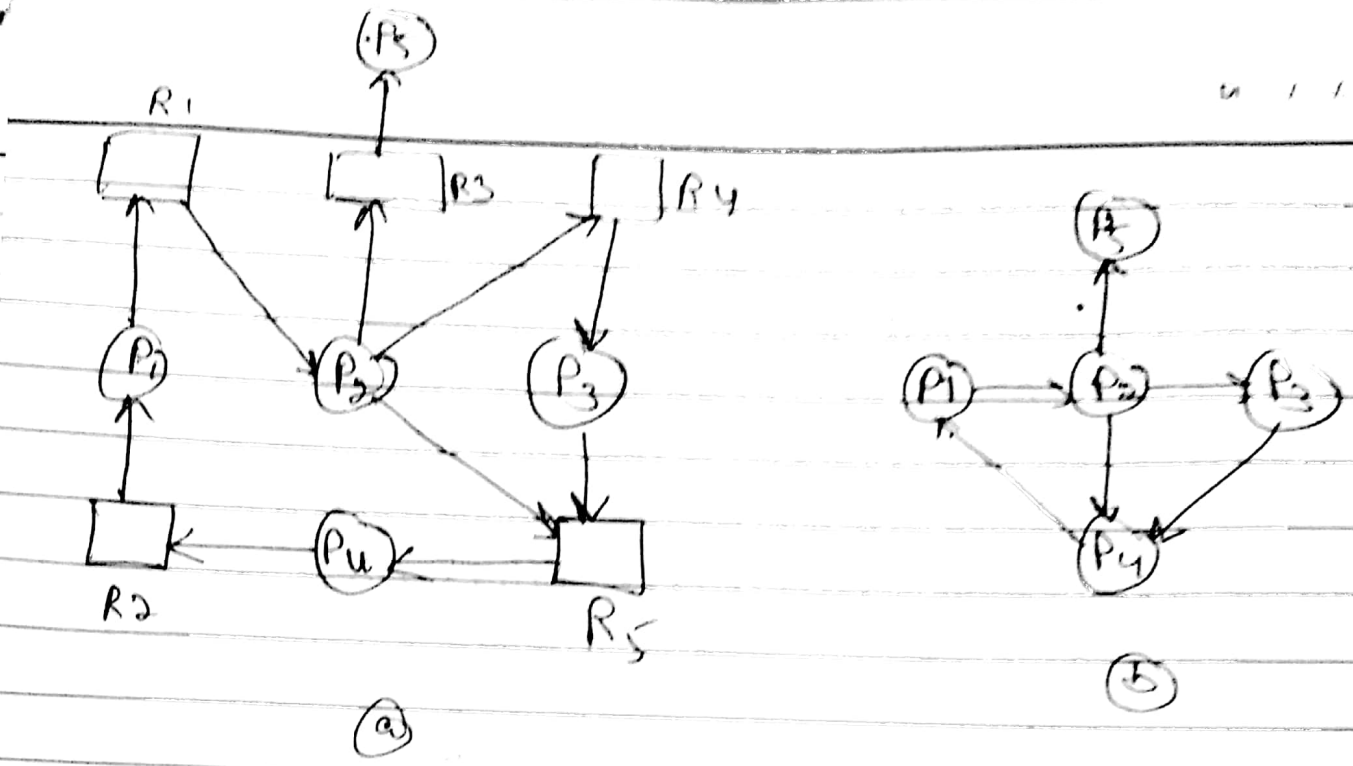
* Single Instance of Each Resource Type :

If all resources have only a single instance, then a deadlock detection algo is used called wait-for graph. We obtain this graph from resource-allocation graph by removing the nodes of resource and collapsing the appropriate edges.

An edge P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i need

$P_i \rightarrow P_j$ $P_i \rightarrow R_q \rightarrow P_j$

$P_i \rightarrow P_j$



An Edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q.

A deadlock exists in the system if and only if the wait-for graph contain cycle.

Several Instances of a Resource Type:

* Available: A vector of length m indicates the no. of available resources of each type.
* Allocation: An nxm matrix defines the no. of resources of each type currently allocated to each process.

Dt...../...../.....

* Request: An n x m matrix indicates the current request of each process. If Request[i,j] = k, then process P_i is requesting k more instances of resource type R_j.

1. Let Work = Available. i = 1, 2, ---n If allocation$_i$ ≠ 0 (In Process) then finish[i] = false otherwise, Finish[i] = true.
2. Find an i such that both
   a. Finish[i] = false
   b. Request$_i$ ≤ Work (Need)

   If no such i exist, go to step 4.
3. Work := Work + Allocation$_i$
   Finish[i] = true
   go to step 2.
4. If finish[i] = false, for some i, $1 \le i \le n$ for some i, $1 \le i \le n$, then the system is in a deadlock state.

Safety Algorithm

Recovery from deadlocks:

When a detection algorithm determines that a deadlocks exists, we have several alternatives to deal with deadlocks. There are two options for breaking a deadlock:

⇒ One is to abort one or more processes to breaks the circular wait.

⇒ Second is to prempt some resources from one or more of the deadlocked processes.

① Process Termination:-

To eliminate deadlocks by aborting a process, we use one of two methods.

a) * Abort all deadlocked processes:-

This method clearly will breaks the deadlock cycle, but at a great expense; these processes may have computed for a long time and result of these computations must be discarded.

b) * Abort one process at a time until the deadlock cycle is eliminated:-

After each process is aborted, a deadlock detection algo

Dt...../...../.....

must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.

If partial termination method is used, then, given a set of deadlocked processes, we must determine which process should be terminated in an attempt to break the deadlock. We should abort those processes the termination of which will incur the minimum cost. Many factors may determine 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 task.
3. How many more resources the process needs in order to complete.
4. How many and what type of resources the process has used.
5. How many processes will need to be terminated
6.

Dt...../...../....

② Resource Preemption :-

To eliminate deadlocks using resource preemption, we prempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

(1) Selecting a victim :-

Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the no. of resources a deadlock process is holding and amount of time a deadlocked process has consumed during its execution.

(2) Rollback :-

If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resources

We must roll back the process to some safe state and restart it from that state.

It is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it.

③ Starvation:

How do we ensure that starvation will not occur? How can we guarantee that resources will not always be preempted from the same process?

It may happen when that the same process is always picked as a victim. As a result, this process never completes its task, a starvation situation occur. We must ensure that a process can be picked as a victim only a finite no. of times.

Process Synchronization

* A cooperating process is that can affect or be affected by other processes executing in the system.
* Co-operating processes may either directly share a logical address space or be allowed to share data only through files.
* Concurrent access to shared data may result in data inconsistency.
* In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share logical address space so that data consistency is maintained.

Process Synchronization is a set of protocols and mechanisms used to preserve system integrity and consistency when concurrent processes share a serially usable resource. A resource is called serially usable resource if it can be used by one process at a time. The state and operation of a serially usable resource may be corrupted when it is manipulated concurrently by more than one process without synchronization. Data access synchronization ensures that shared data do not loose consistency, when they are

updated by several processes. It is implemented by ensuring that accesses to shared data are performed in a mutually exclusive manner.

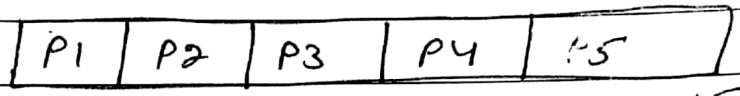
TAT WT

FCFS:-

	AT	BT	CT	TAT	WT
P1	0	4	4	4	0
P2	1	3	7	6	3
P3	2	1	8	6	5
P4	3	2	10	7	5
P5	4	5	15	11	6

$TAT = WT + BT$

Gantt chart



Scheduling Goal criteria

CT → 4

1) Arrival time (AT): The time when process is arrived into a ready state.

2) Burst time: The time required by process for its execution is called burst time for a process

3) Completion time (CT): The time when process is completed is called

4) TAT: The time deffrence b/w CT & arrival time. TAT = CT - AT. or

D1...../...../.....

The Critical-Section Problem:-

* A system with n processes $\{P_0, P_1, \ldots, P_{n-1}\}$. Each process has a segment of code, called a critical section, in which the process may changing common variables, updating a table, writing a file and so on.
* when one process is executing in its critical section no other process is to be allowed to execute in its critical section.
* The execution of critical sections by processes is mutually exclusive in time.
* Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

```
do {
    entry section
    Critical Section
    exit section
    remainder section
} while;
```

waiting time:

5) Diff. b/w TAT & BT
BT inside

WT = TAT - BT

WT = CT - AT - BT

TAT = (17-8) = 9

9 - 10

-1

A solution to the critical-section problem must satisfy the following three requirements:

① Mutual Exclusion:-

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

② Progress:-

If no process is executing in its critical section and some processes wish to enter their critical sections,

Dt...../...../.....

Scheduler:

The selection of process in various queues is carried out by the appropriate scheduler.

① Long-term Scheduler:-

The long term-Scheduler or job scheduler selects processes from mass storage device and loads them into memory for execution.

② Short term Scheduler:

The Short-term scheduler or CPU Scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them.

The difference b/w these two schedulers is the frequency of the execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Because of the brief time b/w execution, the short-term scheduler must be fast.

Dt...../...../....

The long-term scheduler executes much less frequently. The long-term scheduler controls the degree of multiprogramming — the no. of processes in memory. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

In general most process can be described as either

① I/O bound — I/O bound process spends more of its time doing I/O than it spends doing computation.

② CPU bound — CPU bound process spends more of its time doing computation than it spends doing I/O.

The long-term scheduler should a good process mix of I/O bound and CPU bound processes.

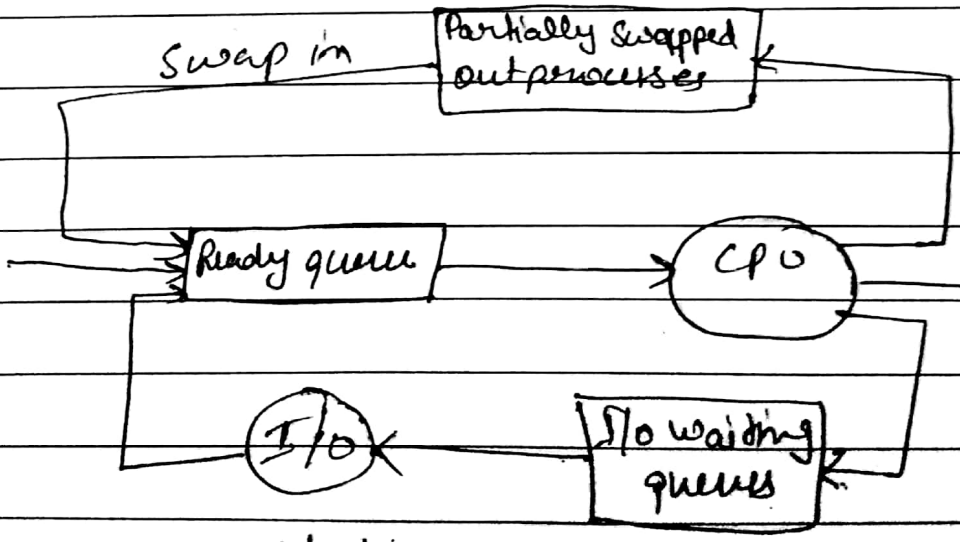
If all the processes are I/O bound the ready queue will always be empty and short-term schedules will have little to do

If the processes are CPU bound, the I/O waiting queue will always be empty, devices will go unused and again the system will be unbalanced. The system with the best performance

will have a combination of CPU-bound and I/O bound process.

(3) In some systems the long term scheduler may be absent. e.g time sharing such as UNIX. In that case there is another scheduler called medium-term scheduler. (Put every new process in memory for short term scheduler). The medium-term scheduler removes processes from memory and thus reduces the degree of multiprogramming. At later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out and is later swapped in by the medium-term scheduler.

Swapping is necessary to improve the process mix or requiring memory to be freed up.



Medium-term Scheduler