Process:

A process is defined as a program in execution. For example computer program written in text file and when we runs (execute) it, it becomes process process execute the code sequentially.

When we execute the same programs multiple times then each copy of the program have different processes.

For e.g.

```
main ( )
{
 int i, prod = 1;
 for ( i = 1; i <= 100, i++)
  {
   prod = prod * i;
   printf ("%d", prod);
  }
```

It is a Program containing one multiplication statement
prod = prod * i
but the process will execute 100 multiplications, one at a time through the for loop.

A program is a passive entity but a process is the active entity

Process States:

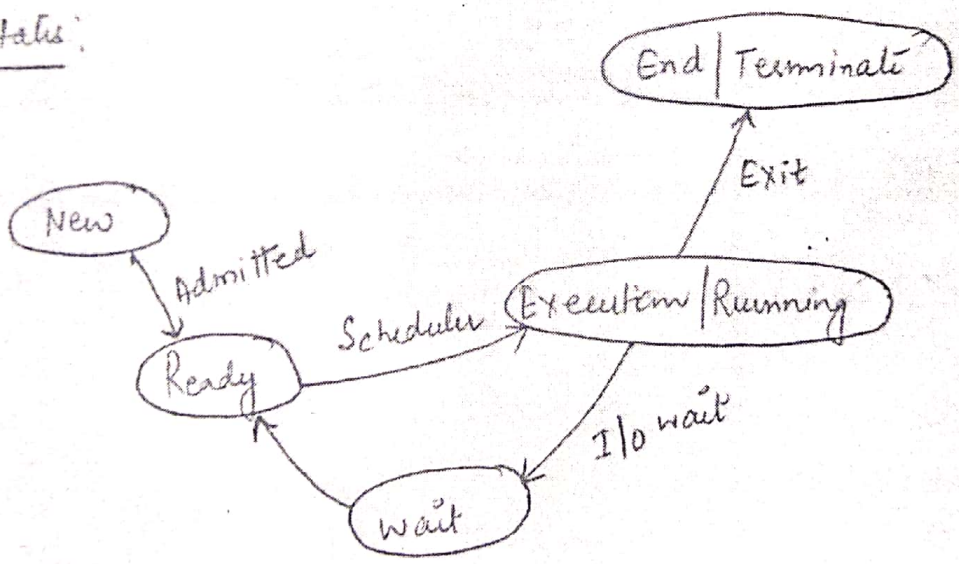


Fig: Process state transition Diagram.

- When a process comes ~~into~~ for the execution, it changes its states from new to terminate
- The state of a process is defined in the path by the current activity of that process.

Each process may be in one of the following state during the time of its execution

(i) New / Creati:
In this state the process is being Created.

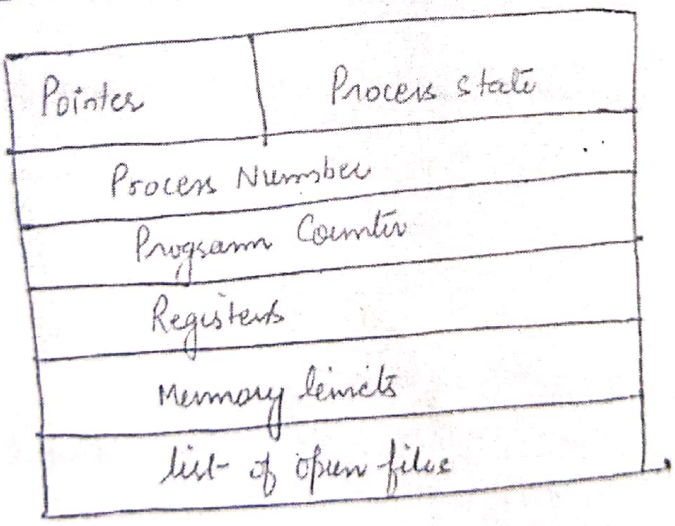
(ii) Ready:
In this state the process is waiting to be assigned to a processor through the scheduler.

(iii) Execution / Running:
In this state, instructions are being executed

(iv) Waiting:
In this state, the process is ~~being~~ waiting for input/output device ~~etc~~ or for processor for execution.

(v) Termination:
In this state, the process has finished its execution.

Process Control:



- In operating System each process is represented by process control block. It is a data structure that physically represent a process in (PCB) memory of a computer.
- It contains many pieces of information associated with a specific process that include the following :-

(i) Pointer:

It is a variable which store the address of the particular process in the memory.

(ii) Process state:

Process may be in one of the following states; new, ready, running, waiting, Termination.

(iii) Process Number: (Identification Tag)

This indicates the process number to identify a specific process.

(iv) Program Counter:

The counter indicates the address of next instruction to be executed for this process.

(v) Registers:

- Registers are the smallest unit of memory or ~~stage~~ storage. They are internal to the CPU.
- They are fastest in terms of accessing the information compared to any other storage.
- Due to limited size of registers, ~~general purpose~~ Many registers can be used at a time such as accumulator, index registers, general purpose registers etc.
- They only store information related to program in execution and their results.
- CPU registers are the temporary storage medium, the information and data delete automatically after termination of the program.
- Their size varies from bytes to kilo bytes. They generally stores the bits in 0 and 1.

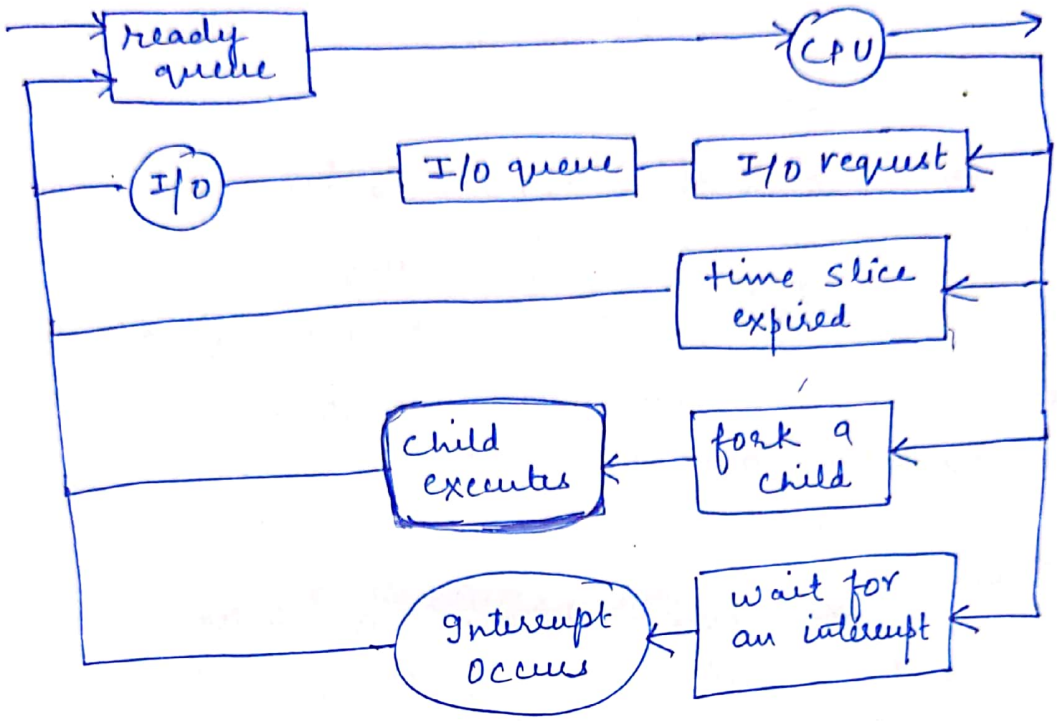
(vi) CPU Scheduling Information:

- This information includes CPU scheduling algorithms (FCFS, SJF, Round Robin, priority based etc.)
- It also includes scheduling queues.

(4)

Process Scheduling: The objective of multiprogramming is to have some processes running all times, and of timesharing to switch among processes so frequently so that user can interact with each program. A uniprocessor system have only one processor, so if multiple process exist, then the rest will wait & need to be rescheduled.

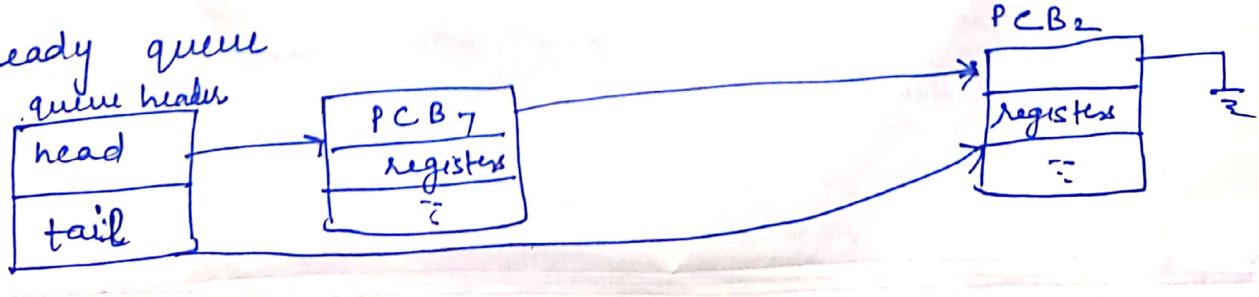
* **Scheduling Queues**



Queueing - diagram representation of process scheduling

→ As processes enters in a system, they are put in job queue. This queue consist of all processes

→ The processes that are residing in main memory & are ready & waiting to execute are stored as linked list & is called ready queue



→ OS has other queues. When a process is allocated the CPU, it executes for a while & eventually quits, is interrupted, or waits for the occurence of a event, such as completion of I/O request.

→ Since the system has many processes, they have to wait for the device. So, the list of processes waiting for I/O device is called a device queue.

A common representation of process scheduling is queueing diagram.

→ Each rectangular box is a queue (two queues ready & device)

→ Each circle represents the resource that serves the queues & arrow indicates flow of processes in the system

A new process is initally put in the ready queue. It waits in the queue until it is selected for execution (or dispatched)

Once the process is assigned to the CPU & is executing, one of several events could occur —

- The process could issue an I/O request, & then be placed in an I/O queue
- The process could create a new subprocess & waits for its termination
- The process could be removed forcibly from the CPU, as a result of an interrupt & be put back in the ready queue.

Schedulers

A process migrates between the various scheduling queue throughout its lifetime. The OS must select, for scheduling purpose, processes from these queues.

The selection procedure is carried out by the appropriate schedulers.

There are three types of scheduler

(1) Long-term scheduler: In batch system, processes are spooled to a mass-storage device (a disk).
→ The long-term scheduler or job scheduler, selects processes from this pool and loads them into memory for execution.
→ It controls degree of multiprogramming ie the number of processes in the memory. If degree of multiprogramming is stable then the avg rate of process creation is equal to the departure rate of processes leaving the system. Most of the processes are either CPU bound process or I/O bound process. The long-term should select a good mix of I/O bound & CPU-bound processes. because if all processes are CPU bound, the I/O waiting queue will almost always be empty & devices will go unused & if all processes are I/O bound, the ready queue will be empty & short term scheduler will have little to do.

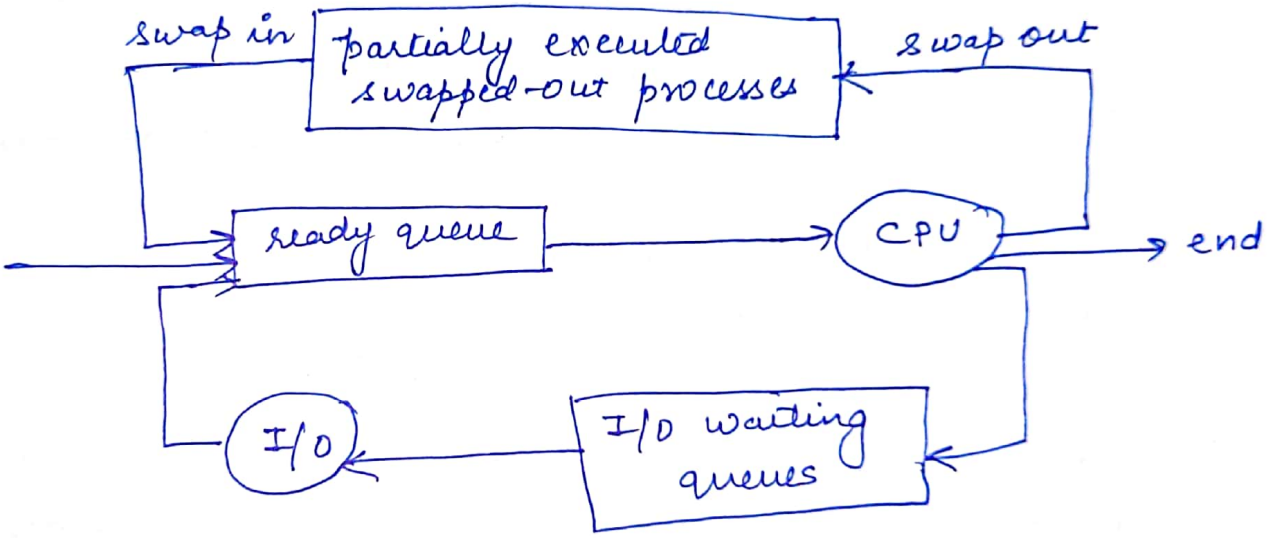
(2) Medium-term scheduler: Some OS, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. Medium term scheduler removes processes from memory & thus reduces degree of multiprogramming

At some later time, the process can be reintroduced into memory & its execution can be continued where it left out. This scheme is called swapping.

The process is swapped out and is later swapped in by medium term scheduler & it helps to improve process mix.

(3) Short-term scheduler

The short term scheduler selects among the processes that are ready to execute & allocates the CPU to one of them. Its execution is fast because it selects a process for the CPU quite frequently ie atleast one in 100ms.



Addition of medium term scheduling to the queueing diagram

Context switch : switching the CPU to another process requires saving the state of the old process & loading the saved state for the new process. This task is known as context switch. It is represented in PCB of a process, it include value of CPU registers, the process state & memory management information.

context switch time is pure overhead & hardware support dependent & is bottleneck for the performance that programmers are using threads to avoid whenever its possible

CPU SCHEDULING

CPU scheduling is the basis of multiprogrammed operating systems.

By switching the CPU among processes, the OS can make the computer more productive.

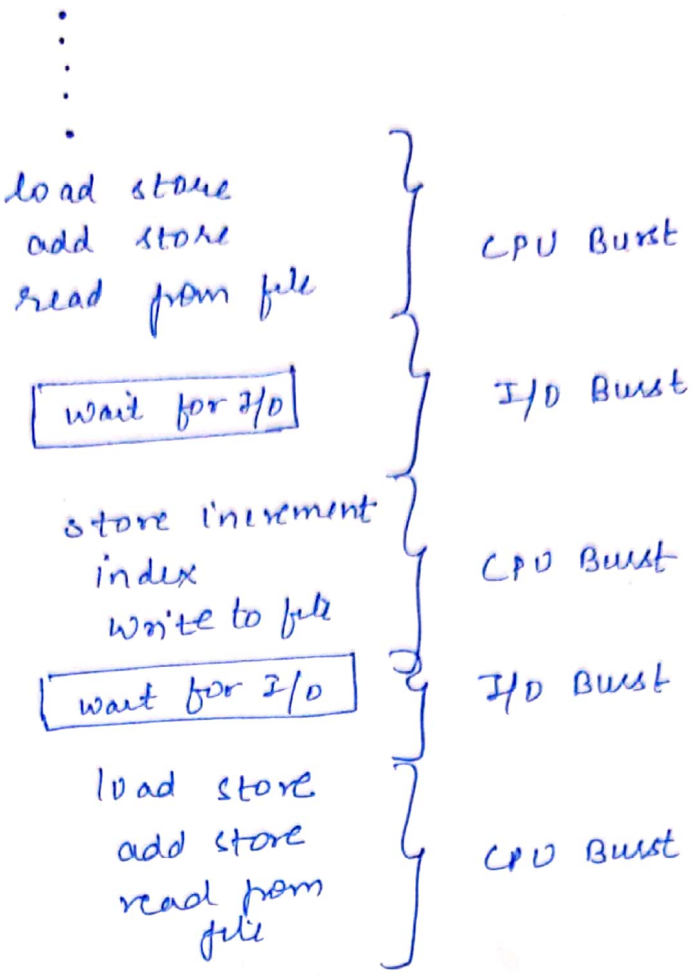
BASIC CONCEPTS

Scheduling is a fundamental operating system function. All computer resources are scheduled before use.

CPU is one of the primary computer resource and thus its scheduling is central to operating system design.

CPU - I/O BURST CYCLE

- The success of CPU scheduling depends on the following observed property of processes.
- Process execution consist of a cycle of CPU execution & I/O wait
- Process alternate between two states
- Process execution begins with a CPU burst followed by an I/O burst, then another CPU burst then another I/O burst & so on.
- The last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.
- An I/O bound program would typically have many very short CPU burst
- A CPU bound program might have a few very long CPU burst



Alternate sequence of CPU & I/O burst

CPU SCHEDULER

- When CPU becomes idle, the OS must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the short-term scheduler (or CPU scheduler)
- The scheduler selects among the processes in memory that are ready to execute & allocates the CPU to one of them
- A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.
- The records in the queues are generally process control blocks (PCB) of the processes.

Premptive Scheduling

CPU scheduling decisions take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example I/O request, or invocation of wait for the termination of one of the child process
2. When a process switches from the ~~beginning~~ running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from waiting state to the ready state (for example consider I/O)
4. When a process terminates.

In circumstances 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, in a circumstance 2 and 3

When scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is <u>non-premptive</u>, otherwise scheduling scheme is <u>premptive</u>

Dispatcher

Dispatcher is a module that gives control of the CPU to the process selected by the short term scheduler. Their function involves:

- switching context
- switching to user mode
- Jumping to proper location in the user program to restart that program

The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

Scheduling Criteria

There are many criteria that are used to compare various CPU-scheduling algorithms. The characteristics used for comparison can make a substantial difference in determination of the best algorithm. The criteria include:

- CPU utilization: CPU should be as busy as possible. CPU utilization may range from 0 to 100 percent. In real time system, it should range from 40 percent (for lightly loaded) to 90 percent (for a heavily used system)
- Throughput: It is described as the no. of processes completed per time unit. For long processes, this rate may be 1 process per hour, for short transactions, it can be 10 processes per second.
- Turnaround time: The interval from the time of submission of a process to the time of completion is the turnaround time

> Turnaround time = Waiting (to get in memory + for I/O + in ready queue) + executing on the CPU

- waiting time: It is the sum of time waiting in the ready queue.
- Response time: The time from the submission of request until the first response is produced.

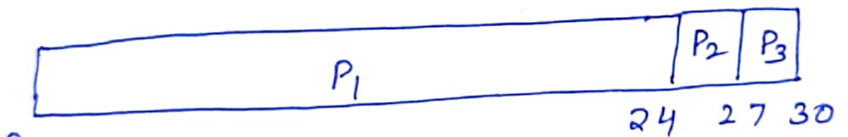
SCHEDULING ALGORITHMS

First-come, First Served Scheduling (FCFS)

- The simplest scheduling algorithm is the First come - first served (FCFS) algorithm.
- In this, the process that requests the CPU first is allocated the CPU first
- Implementation of the FCFS policy is managed by FIFO queue.
- The average waiting time is quite long in FCFS

Process	Burst Time
P_1	24
P_2	3
P_3	3

GANTT CHART



Waiting Time (P_1) = 0
Waiting Time (P_2) = 24
Waiting Time (P_3) = 27

$$\text{Average Waiting Time} = \frac{\text{Waiting time of all processes}}{\text{Number of processes}}$$

So, Average waiting Time $= \frac{0+24+27}{3} = \frac{51}{3} = 17$

Turnaround Time (P_1) = WT (P_1) + Burst Time (P_1) = 0 + 24 = 24
Turnaround Time (P_2) = WT (P_2) + BT (P_2) = 24 + 3 = 27
Turnaround Time (P_3) = WT (P_3) + BT (P_3) = 27 + 3 = 30

$$\text{Average Turnaround Time} = \frac{\text{Turnaround Time of all processes}}{\text{Number of processes}}$$

So,

Average Turnaround Time $= \frac{24+27+30}{3}$

$= \frac{81}{3} = 27.$

Shortest Job first Scheduling: SJF

↳ In SJF works on the process with the shortest burst time or duration first. This is the best approach to minimize waiting time.

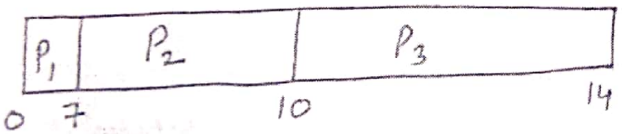
↳ It is of two types:
1. Non preemptive
2. Pre-emptive

Drawback:
the burst time / duration time of the processes should be known to the processor in advance, which is not possible (feasible)

Non-preemptive Shortest-Job first Scheduling:

Process	Burst Time	Arrival Time	Completion Time	TAT	WT
P_1	7	0	7	7-0=7	7-7=0
P_2	3	1	10	10-1=9	9-3=6
P_3	4	3	14	14-3=11	14-4=10

First we need to draw the Gantt chart for non preemptive SJF



Average waiting time = $\frac{\text{Total waiting time of all processes}}{\text{no. of processes.}}$

$= (0+7+10)/3 = 17/3 = 5.6$ ms

Average Turnaround time = $\frac{\text{Total Turnaround time of all proc}}{\text{No. of processes.}}$

$= \frac{7+9+11}{3} = \frac{27}{3} = 9$ ms.

Note: Here all the processes are arrived at different time so when P_1 arrived no process was in ready queue so we assign the CPU to P_1 (no matter what is the burst time). When the P_1 completed its execution then we have two more process in ready queue P_2 and P_3 So scheduler select the process having lowest burst time among P_2 and P_3. Therefore P_2 will execute next and then P_3 will executed.

Shortest Remaining Time First (SRTF) Scheduling:

↳ Preemptive version of Shortest job first (SJF) scheduling is known as SRTF scheduling.

↳ ~~With the help~~ In SRTF, the process having the smallest amount of time remaining until completion is selected first.

↳ So in SRTF, the process are scheduled according to the shortest remaining time.

↳ SRTF have more overheads than SJF because in SRTF Operating system is required frequently to monitor the CPU time of the jobs in the ready queue and to perform context switching.

↳ At the arrival of every process, the short term scheduler schedules the processes with the minimum remaining burst time among the list of available processes and the running process.

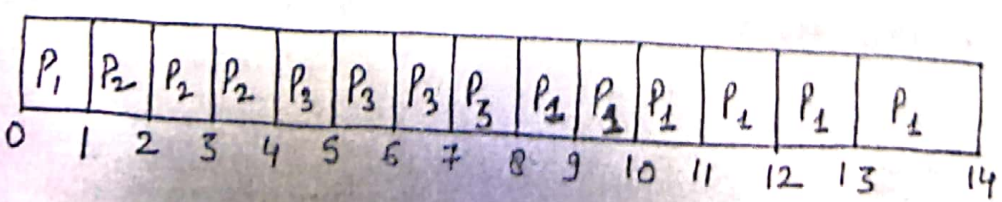
↳ When all processes are available in ready queue, no preemption will be done.

↳ The context of the process is saved in the PCB, when the process is removed from the execution and next process is scheduled. PCB is accessed on the next execution of this process.

Example:

Process	Burst time	Arrival Time	completion Time	TAT = CT-AT	WT = TAT-BT
P_1	7	0	14	14-0 = 14	14-7 = 7
P_2	3	1	4	4-1 = 3	3-3 = 0
P_3	4	3	8	8-3 = 5	5-4 = 1

First step is to make the Gantt chart for SRTF:



Round Robin Scheduling:

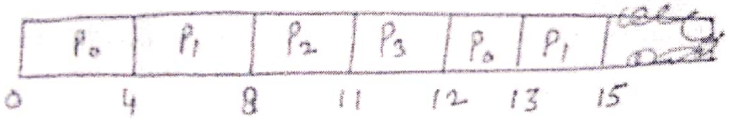
- Round Robin is the preemptive process scheduling algo.
- Each process is provided a fix time to execute, it is called a quantum or time slice.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save state of preempted processes.

Example:

Process	Arrival Time AT	Burst time BT	Completion Time CT	TAT CT-AT	WT TAT-BT
P_0	0	5	13	13-0=13	13-5=8
P_1	1	6	15	15-1=14	14-6=8
P_2	2	3	11	11-2=9	9-3=6
P_3	3	1	12	12-3=9	9-1=8

Time Quantum or Time slice = 4 units, find Avg W.T & Avg TAT

Sol:



Waiting time for P_0 = 8 Unit

" " " P_1 = 8 Unit

" " " P_2 = 6 Unit

" " " P_3 = 8 Unit

$$\text{Average waiting time} = \frac{\text{total waiting time}}{\text{Total no. of Processes}} = \frac{(8+8+6+8)}{4} = \frac{30}{4} = 7$$

Turnaround time for Process P_0 = 13 Unit

" " " P_1 = 14 Unit

" " " P_2 = 9 Unit

" " " P_3 = 9 Unit

$$\text{Average Turnaround Time} = \frac{\text{total Turnaround Time}}{\text{total no. of Process}} = \frac{(13+14+9+9)}{4}$$

$$= \frac{45}{4} = 11.2 \text{ Unit}$$

Average waiting Time = $\frac{\text{Waiting time of all process}}{\text{No. of process}}$

Avg waiting Time = $(7+0+1)/3 = 8/3 = 2.66$ ms.

Average Turnaround Time = $\frac{\text{Turn around time of all process}}{\text{No. of process}}$

Avg Turn around Time = $(14+3+5)/3 = \frac{22}{3} = 7.3$ ms.

Example: Consider the following process

Process	Arrival Time	Burst Time	Priority
P_1	0	8	3
P_2	1	5	1
P_3	2	16	2
P_4	3	8	4

Draw Gantt chart and find average waiting time, average Turnaround Ti

(a) SRTF Scheduling

(b) Round Robin Scheduling (time quantum = 4)

Priority Scheduling:

↳ In Priority scheduling, priority is associated with each process and the CPU is scheduled to process with the highest priority

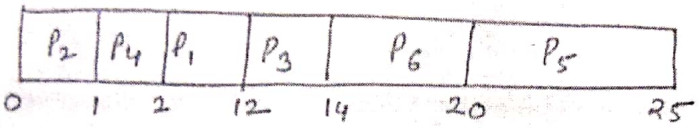
↳ If two jobs are having same priority then processes are scheduled in FCFS order.

Example:

Process	Burst Time	Priority	Completion Time	TAT	WT
P_1	10	3	12	12	2
P_2	1	1	1	1	0
P_3	2	4	14	14	12
P_4	1	2	2	2	1
P_5	5	6	25	25	20
P_6	6	5	20	20	14

Draw Gantt chart and find the average W.T and Avg TAT.

Sol:



Gantt chart for priority scheduling

Avg waiting time = $(2+0+12+1+20+14)/6 = 49/6 = 8.16$ ms

Avg Turn around time = $(12+1+14+2+25+20)/6 = 74/6 = 12.3$

↳ Priority scheduling can be either preemptive or non-preemp ie when a process arrives at the ready queue, its priority is compared with the priority of the current running process and if the priority of the arrived process is higher than in preemptive scheduling the current process will be prempt

↳ while in non-preemptive the arrived process will be simply put at the head of the ready queue.

↳ A major problem with priority scheduling is indefinite blocking (or starvation) ie a low priority process will wait for CPU forever & never gets CPU

↳ Solution of Starvation or indefinite blocking is aging. Aging is a technique of gradually increasing priority of processes that wait in the system for a long time.

Multilevel Queue scheduling

→ In this processes are classified into different groups ie foreground (interactive) processes & background (batch) processes

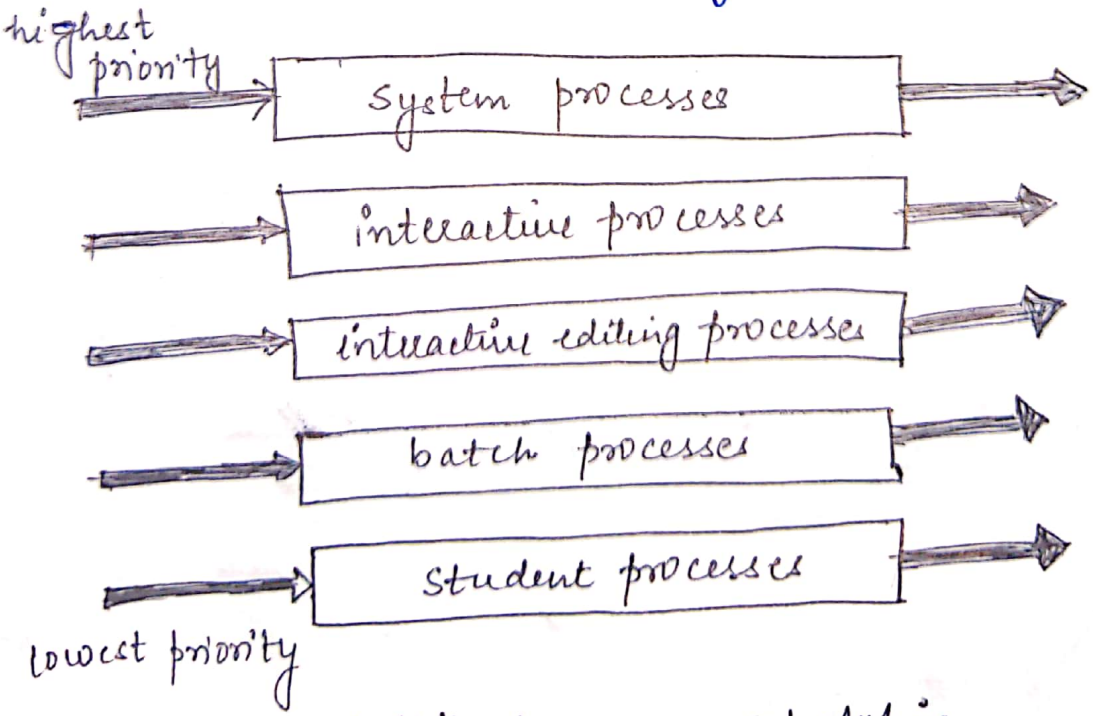
→ Foreground processes may have priority (externally) over background processes

→ A multilevel queue-scheduling algorithm partitions the ready-queue into separate queues.

→ The processes are assigned to one queue permanently on some property of the process such as memory size, process priority, process size

→ Each queue has its own scheduling algorithm

→ The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.



Multilevel queue scheduling

→ In addition, there must be scheduling among the queues, which is implemented as fixed-priority preemptive scheduling.

→ Multilevel queue scheduling algorithm has five queues namely:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

→ No process in batch queue can run unless the queue for system process, Interactive processes, interactive editing processes were all empty.

→ If an interactive editing process enters in the ready queue while a batch process is running, the batch process will be prempted.

→ Another possibility can be, to give time slice between the queue ie a certain portion of CPU will be given to foreground processes (ie around 80%) & rest to background processes (20%).

Multilevel Feeback Queue Scheduling

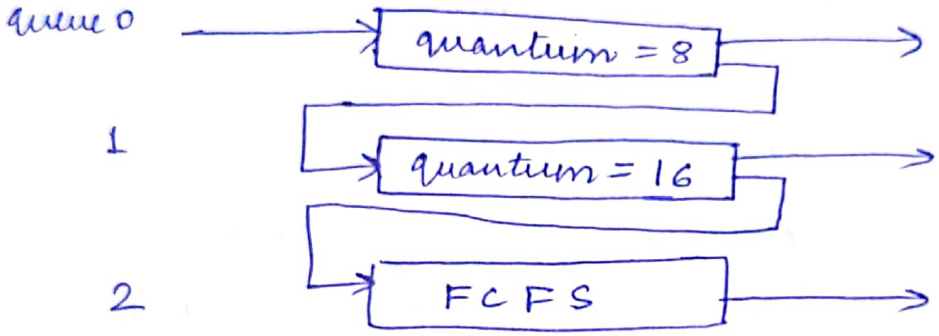
→ The disadvantage of multilevel queue scheduling is that the processes cannot move from one queue to another & thus being inflexible

↳ Multilevel feedback queue scheduling, allows process to move between queues.

↳ The idea is to separate processes with different CPU-burst characteristics.

↳ If a process uses too much CPU time, it will be moved to lower priority queue. This scheme leaves I/O bound and interactive processes in the higher priority queues.

↳ Similarly 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



Multilevel feedback queues

↳ The scheduler first executes all processes in queue 0
↳ then in queue 1 & then in queue 2
A process that arrives in queue 1 will prempt a process in queue 2 & a process in queue 0 can prempt a process in queue 1

↳ When a process enters in a ready queue it should be in queue 0. and time quantum of 8 ms will be given, it will be moved to tail of queue 1

↳ If queue 0 is empty, the process at the head of queue 1 is given time quantum of 16 ms.

↳ If still, it not complete, it is prempted & is put into queue 2

↳ Processes in queue 2 are run on an FCFS basis, only when queues 0 & 1 are empty.

A multilevel feedback queue scheduler is defined by the following parameters:

- The no. 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 service.

Multiprocessor Scheduling

↳ When multiple processors are available, then the scheduling gets more complicated because there is more than one CPU which must be kept busy & in effective use at all times.

↳ Load sharing revolves around balancing the load between multiple processors

↳ Multiprocessor can be homogenous (all the same kind of CPU) or heterogenous (different kind of CPU)

↳ If identical processors are available, load sharing can occur. If separate queues are provided then one queue can be empty & other very busy

↳ To prevent this, we use common ready queue. All processes go into one queue & are scheduled onto any available processor.

↳ In above scheme two scheduling schemes can be used – ① self scheduling i.e each processor is self-scheduled & examines common ready queue & selects a process to execute. But must programmed so that two processors do not choose the same process.

↳ ② In this approach avoid the above problem i.e not there as it appoints one processor as scheduler for the other processors, thus creating master slave structure

↳ Some system using this structure one step further, by having all scheduling decisions, I/O processing & other system activities handled by one single processor- the master server i.e uses asymmetric multiprocessing

↳ It is not efficient as I/O bound process may bottleneck on the one CPU that is performing all the operations.

3. Deadlocks

In a multiprogramming environment several processes may compete for a finite number of resources. A process request resources; if the resource is available at that time a process enters thewait state. Waiting process may never change its state because the resources requested are held by other waiting process. This situation is known as deadlock.

3.1 System Model:

A system consists of a finite number of resources to be distributed among a number of competingprocesses. The resources are partitioned into several types each of which consists of a number of identical instances. A process may utilized a resource in the following sequence :

- **Request:** In this state one can request a resource.
- **Use:** In this state the process operates on the resource.
- **Release:** In this state the process releases the resources.

3.2 Deadlock Characteristics: In a deadlock, process never finish executing and system resources are tied up preventing other jobs from starting.

3.2.1 →Necessary Conditions:

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

1.Mutual Exclusion: At a time only one process can use the resources. If another process requests that resource, requesting process must wait until the resource has been released.

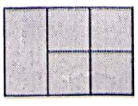
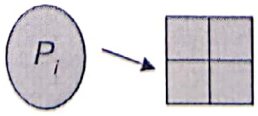
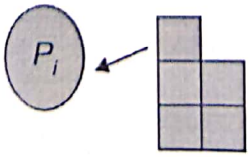
2.Hold and wait: A process must be holding at least one resource and waiting to additionalresource that is currently held by other processes.

3.No Preemption: Resources allocated to a process can't be forcibly taken out from it unless it releases that resource after completing the task.

4.Circular Wait: A set $\{P_0, P_1, \ldots\ldots P_n\}$ of waiting state/ process must exists such that P_0 iswaiting for a resource that is held by P_1, P_1 is waiting for the resource that is held by P_2 ..... $P_{(n-1)}$ is waiting for the resource that is held by P_n and P_n is waiting for the resources that is held by P_0.

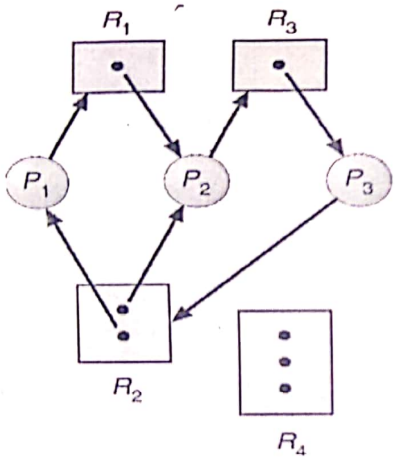
3.2.2 →Resource Allocation Graph:

Deadlock can be described more clearly by directed graph which is called system resource allocation graph. The graph consists of a set of vertices 'V' and a set of edges 'E'. The set of vertices 'V' is partitioned into two different types of nodes such as $P = \{P_1, P_2, \ldots\ldots P_n\}$, the set of all the active processes in the system and $R = \{R_1, R_2, \ldots\ldots R_m\}$, the set of all the resource type in the system. A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$. It signifies that process P_i is an instance of resource type R_j and waits for that resource. A directed edge from resource type R_j to the process P_i which signifies that an instance of resource type R_j has been allocated to process P_i. Adirected edge $P_i \rightarrow R_j$ is called as request edge and $R_j \rightarrow P_i$ is called as assigned edge.

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 
- P_i is holding an instance of R_j 

When a process P_i requests an instance of resource type R_j then a request edge is inserted as resource allocation graph. When this request can be fulfilled, the request edge is transformed to an assignment edge. When the process no longer needs access to the resource it releases the resource and as a result the assignment edge is deleted. The resource allocation graph shown in below figurehas the following situation.

- The sets P, R, E
- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$ The resource instances are
  - Resource R_1 has one instance
  - Resource R_2 has two instances.
  - Resource R_3 has one instance
  - Resource R_4 has three instances.

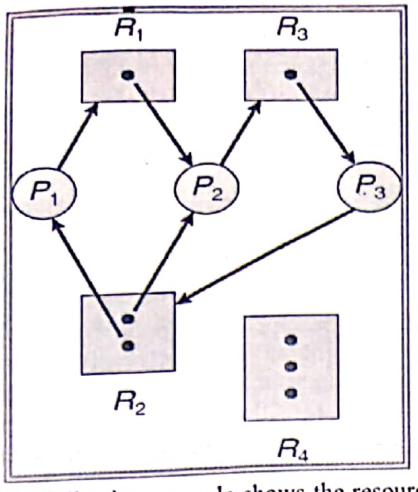


The process states are:

- Process P_1 is holding an instance of R_2 and waiting for an instance of R_1.
- Process P_2 is holding an instance of R_1 and R_2 and waiting for an instance R_3.
- Process P_3 is holding an instance of R_3.

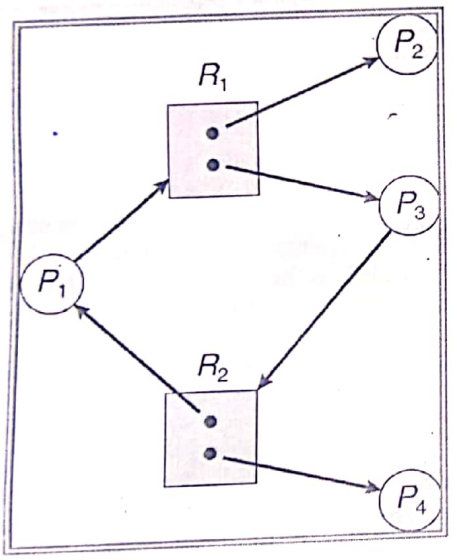
The following example shows the resource allocation graph with a deadlock.

- P1 -> R1 -> P2 -> R3 -> P3 -> R2 -> P1
- P2 -> R3 -> P3 -> R2 -> P2



The following example shows the resource allocation graph with a cycle but no deadlock.

- P1 -> R1 -> P3 -> R2 -> P1
- No deadlock
- P4 may release its instance of resource R2
- Then it can be allocated to P3



3.3 Methods for Handling Deadlocks

The problem of deadlock can deal with the following 3 ways.

- We can use a protocol to prevent or avoid deadlock ensuring that the system will never enterto a deadlock state.
- We can allow the system to enter a deadlock state, detect it and recover.
- We can ignore the problem all together.

To ensure that deadlock never occur the system can use either a deadlock prevention or deadlock avoidance scheme.

3.3.1 Deadlock Prevention:

Deadlock prevention is a set of methods for ensuring that at least one of these necessary conditionscannot hold.

- **Mutual Exclusion:** The mutual exclusion condition holds for non sharable. The example is aprinter cannot be simultaneously shared by several processes. Sharable resources do not require mutual exclusive access

and thus cannot be involved in a dead lock. The example is read only files which are in sharing condition. If several processes attempt to open the read only file at the same time they can be guaranteed simultaneous access.

- **Hold and wait:**To ensure that the hold and wait condition never occurs in the system, we must guaranty that whenever a process requests a resource it does not hold any other resources. There are two protocols to handle these problems such as one protocol that can be used requires each process to request and be allocated all its resources before it begins execution. The other protocol allows a process to request resources only when the process has no resource. These protocols have two main disadvantages. First, resource utilization may below, since many of the resources may be allocated but unused for a long period. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some otherprocess.
- **No Preemption:** To ensure that this condition does not hold, a protocol is used. If a processis holding some resources and request another resource that cannot be immediately allocated to it. The preempted one added to a list of resources for which the process is waiting. The process will restart only when it can regain its old resources, as well as the new ones that it is requesting. Alternatively if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If theresources are not either available or held by a waiting process, the requesting process must wait.
- **Circular Wait:**We can ensure that this condition never holds by ordering of all resource typeand to require that each process requests resource in an increasing order of enumeration.
  Let R= {R_1, R_2, ...R_n}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. Formally, we define a one to one function F: R □ N, whereN is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives and printers, then the function F might be defined as follows:
  F (Tape Drive) = 1,F
  (Disk Drive) = 5, F
  (Printer) = 12.
  We can now consider the following protocol to prevent deadlocks: Each process can requestresources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type, say R_i. After that, the process can request instances of resource type R_j if and only if F (R_j) > F (R_i). If several instances of the same resource type are needed, defined previously, a process that wants to use the tape drive andprinter at the same time must first request the tape drive and then request the printer.

3.3.2 Deadlock Avoidance

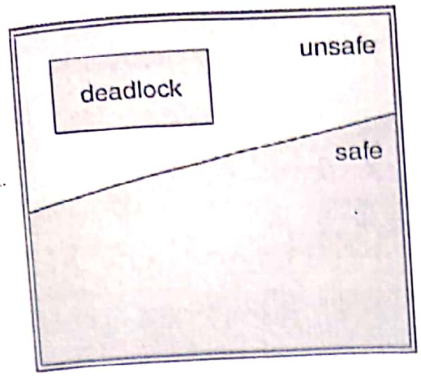
Requires additional information about how resources are to be used.Simplest and most useful modelrequires that each process declare the maximum number of resources of each type that it may need.The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State

When a process requests an available resource, system must decide if immediate allocation leaves thesystem in a safe state.Systems are in safe state if there exists a safe sequence of all process. A sequence <P_1, P_2, ..., P_n> of ALL the processes is the system such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, withj <i.That is:

- If P_i resource needs are not immediately available, then P_i can wait until all $P_j$have finished.
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, andterminate.
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on.
- If system is in safe state => No deadlock
- If system in not in safe state => possibility of deadlock
- OS cannot prevent processes from requesting resources in a sequence that leads to deadlock

- Avoidance => ensue that system will never enter an unsafe state, prevent getting into deadlock



Example:

Available resources = 3

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

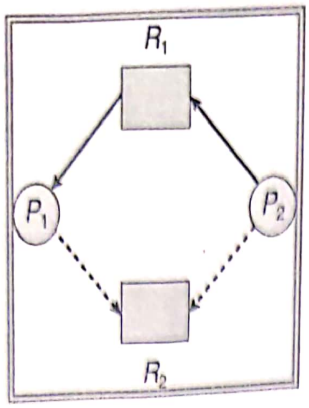
- Suppose processes P0, P1, and P2 share 12 magnetic tape drives
- Currently 9 drives are held among the processes and 3 are available
- Question: Is this system currently in a safe state?
- Answer: Yes!
  - Safe Sequence: <P1, P0, P2>

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

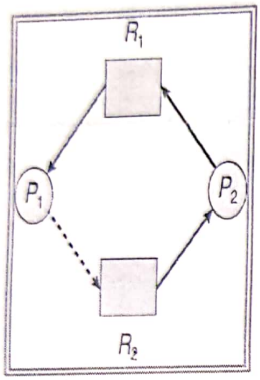
- Suppose process P2 requests and is allocated 1 more tape drive.
- Question: Is the resulting state still safe?
- Answer: No! Because there does not exist a safe sequence anymore.
  - Only P1 can be allocated its maximum needs.
  - IFP0 and P2 request 5 more drives and 6 more drives, respectively, then the resultingstate will be deadlocked.

Resource Allocation Graph Algorithm

In this graph a new type of edge has been introduced is known as claim edge. Claim edge $P_i \Rightarrow R_j$ indicates that process P_j may request resource R_j; represented by a dashed line.Claim edge converts to request edge when a process requests a resource.Request edge converted to an assignment edge when the resource is allocated to the process.When a resource is released by a process, assignment edge reconverts to a claim edge.Resources must be claimed a priori in the system.



- P2 requesting R1, but R1 is already allocated to P1.
- Both processes have a claim on resource R2
- What happens if P2 now requests resource R2?



- Cannot allocate resource R2 to process P2
- Why? Because resulting state is unsafe
  - P1 could request R2, thereby creating deadlock!

Use only when there is a single instance of each resource type

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge doesnot result in the formation of a cycle in the resource allocation graph.
- Here we check for safety by using cycle-detection algorithm.

Banker's Algorithm

This algorithm can be used in banking system to ensure that the bank never allocates all its availablecash such that it can no longer satisfy the needs of all its customer. This algorithm is applicable to a system with multiple instances of each resource type. When a new process enter in to 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. Several data structure must be maintained to implement the banker's algorithm.
Let,

- n = number of processes
- m = number of resources types

- **Available:** Vector of length m. If Available[j] = k, there are k instances of resource type R_j available.
- **Max:** n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j.
- **Allocation:** n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.
- **Need:** n x m matrix. If Need[i,j] = k, then P_i may need k more instances of R_j to complete its task.
  Need [i,j] = Max[i,j] – Allocation [i,j].

Safety Algorithm

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

2. Find an 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 ~~Allocation Algorithm~~ Request Algorithm : whether request can be safely granted.

Let $Request_i$ be the request vector 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:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows: Available =
   Available – Request;
   $Allocation_i$ = $Allocation_i$ + $Request_i$;
   $Need_i$ = $Need_i$ – $Request_i$;
   - If safe ⇒ the resources are allocated to Pi.
   - If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Example:

- 5 processes P_0 through P_4;
- 3 resource types:
  - A (10 instances), B (5instances), and C (7 instances).
- Snapshot at time T_0:

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

- The content of the matrix Need is defined to be Max – Allocation. 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

- The system is in a safe state since the sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies safety criteria. If P_1 requests (1, 0, 2) What will happen? check request$_1$ < need$_1$ i.e. 1,0,2 < 1,2,2 = true
- Check that Request $\leq$ Available (that is, (1,0,2) $\leq$ (3,3,2) $\equiv$ true.

	Allocation	Need	Available
		A B C	A B C A B C
P_0		0 1 0	7 4 3 2 3 0
P_1		3 0 2	0 2 0
P_2		3 0 1	6 0 0
P_3		2 1 1	0 1 1
P_4		0 0 2	4 3 1

Available = Available – Request$_1$
Allocation = Allocation$_1$ + Request$_1$
Need$_1$ = Need$_1$ – Request$_1$

- Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted? –NO
- Can request for (0,2,0) by P_0 be granted? –NO (Results Unsafe)

3.3.3 Deadlock Detection

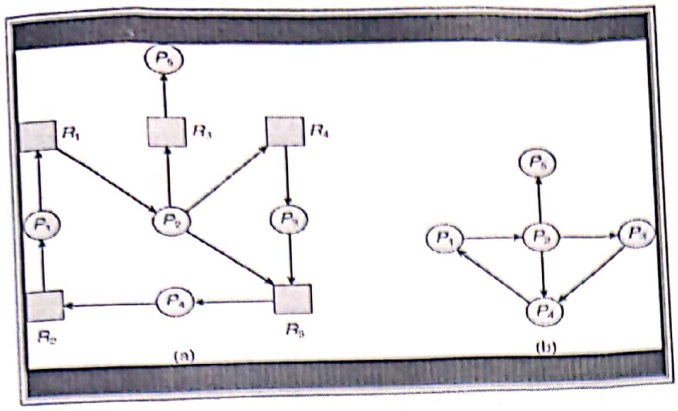
If a system doesn't employ either a deadlock prevention or deadlock avoidance, then deadlocksituation may occur. In this environment the system must provide

- An algorithm to recover from the deadlock.
- An algorithm to remove the deadlock is applied either to a system which pertains single in instance each resource type or a system which pertains several instances of a resource type.

Single Instance of each Resource type

If all resources only a single instance then we can define a deadlock detection algorithm which uses anew form of resource allocation graph called "Wait for graph". We obtain this graph from the resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges. The below figure describes the resource allocation graph and corresponding wait for graph.

- $P_i \rightarrow P_j$ (P_i is waiting for P_j to release a resource that P_i needs)
- $P_i \rightarrow P_j$ exist if and only if RAG contains 2 edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q



a.Resource-AllocationGraph

b.Corresponding wait-for graph

Several Instances of a Resource type

The wait for graph scheme is not applicable to a resource allocation system with multiple instances of reach resource type. For this case the algorithm employs several data structures which are similarto those used in the banker's algorithm like available, allocation and request.

- **Available:** A vector of length m indicates the number of available resources of each type.
- **Allocation:** An n x m.matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An n x m matrix indicates the current request of each process. If Request [i_j] = k,then process P_i is requesting k more instances of resource type. R_j.

1. Let Work and Finish be vectors of length m and n, respectively Initialize:

(a) Work = Available

(b) For i = 1,2, …, n, if Allocation$_i$!= 0,
then
(c) Finish[i] = false; otherwise, Finish[i]
= true.

2. Find an index i such that both:

(a) Finish[i] == false

(b) Request$_i$<=Work

If no such i exists, go to step 4.

3. Work=Work
+Allocation
Finish [i] = true
Go to step 2

4. If Finish [i] = false, for some i, 1<=i<= n, then the system is in a deadlock state. Moreover, if Finish[i]<=false, then process Pi is deadlocked.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

Process Termination:

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, thesystem reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes:** This method clearly will break the deadlock cycle, but at agreat expense; these processes may have computed for a long time, and the results of these partial

computations must be discarded and probably recomputed later.

- **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted, a deadlock detection algorithmmust be invoked to determine whether any processes are still deadlocked.

Resource Preemption:

To eliminate deadlocks using resource preemption, we successively preempt 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.

- **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 numbers of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.
- **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 resource. We must rollback the process to some safe state, and restart it from that state.
- **Starvation:** In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a small finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Example: Banker's Algorithm: consid system with five processes P_0 to P_4 & three resource of type A, B, C. Suppose at time to following snapshot of the system has been taken

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Q2 Is system is in safe state? If yes, what is the safe sequence?

Step 1 of safety algo

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	F	F	F	F	F

Step-2

For i = 0

Need$_0$ = 7, 4, 3

Finish[0] is false & Need > Work

7 4 3 > 3 3 2

So P_0 must wait

Step-2

For i = 1

Need$_1$ = 1, 2, 2

Finish[1] is false & need < work

So P_1 must be kept in safe sequence

Work = Work + Allocation$_1$ = 3 3 2 + 2 0 0 = 5 3 2 Step 3

Q1 what will be need matrix

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

Process	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

Finish = | 0 | 1 | 2 | 3 | 4 |
| false | T | F | F | F |

Step 2

For $i=2$, $Need_2 = 6,0,0$

Finish[2] is false & $Need_2 > work$

$6,0,0 > 5,3,2$

So, P_2 must wait

Step 2

For $i=3$, $Need_3 = 0,1,1$

Finish[3] = false & $Need_3 < work$

$0,1,1 \; , \; 5,3,2$

So, P_3 must be kept in safe sequence

Step 3

$work = work + allocation_3$

$work = 5,3,2 + 2,1,1$

$work = 7,4,3$

Finish = | F | T | F | T | F |

Step 2

For $i=4$

$Need = 4,3,1$

Finish[4] = F & $Need < work$

$4,3,1 < 7,4,3$

So P_4 must be kept in safe sequence

Step 3

$work = work + allocation_4$

$= 7,4,3 + 0,0,2$

work = | A | B | C |
| 7 | 4 | 5 |

Finish = | 0 | 1 | 2 | 3 | 4 |
| F | T | F | T | T |

Step 2

For $i=0$

$Need = 7,4,3$

Finish[0] is false & $Need < work$ ($7,4,3 < 7,4,5$)

So, P_0 must be kept in safe sequence

Step - 3

$work = work + Allocation$

$7,4,5 + 0,1,0$

work = | 7 | 5 | 5 |

Finish = | 0 | 1 | 2 | 3 | 4 |
| true | true | false | true | true |

Step 2

For $i=2$

$Need = 6,0,0$

Finish[2] is false & $Need_2 \leq work$ ($6,0,0 \leq 7,5,5$)

So P_2 must be kept in safe sequence

Step 3

$work = work + Allocation_2$

$work = 7,5,5 + 3,0,2$

work = | 10 | 5 | 7 |

Finish = | 0 | 1 | 2 | 3 | 4 |
| T | T | T | T | T |

Step 4

Finish[i] = true for $0 \leq i \leq n$

Hence the system is in safe state

P_1, P_3, P_4, P_0, P_2

This is the safe sequence