Process Scheduling

- Scheduling objectives –

  - Maximize the number of interactive users within acceptable response times.
  - Achieve a balance between response and utilization.
  - Avoid indefinite postponement and enforce priorities.
  - It also should give reference to the processes holding the key resources.
  - Maximize throughput
  - Enforce priorities
  - Give better service to processes that have desirable behavior patterns.
  - Degrade gracefully under heavy loads.

- Basic concepts –

  - The idea of multiprogramming is relatively simple. A process is executed until it must wait, typically for the completion of some I/O requests.
  - In a simple computer system, the CPU would then sit ideal; all this waiting time is wasted.
  - With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait the OS takes the CPU away from that process and given the CPU to another process. This pattern continues. Scheduling is a fundamental OS functions. Almost all computer resources are scheduled before use.
  - The CPU is of course one of the primary computer resources.
  - Thus, its scheduling is central to OS assign.

- CPU I/O burst cycle –

The success of CPU scheduling depends on the following observed property of process:

  - Process execution consists of a cycle of CPU execution and I/O wait processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, then another CPU burst, then another I/O burst and so on. Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.
  - The duration of their CPU burst has been extensively measured.
  - Although they vary greatly by process and by computer, they tend to have a frequency curve.
  - The curve is generally characterized as exponential or hyper exponential, with many short CPU bursts and a few long CPU bursts.
  - An I/O bound program would typically have many short CPU bursts.
  - This distribution can help us select an appropriate CPU scheduling algorithm.

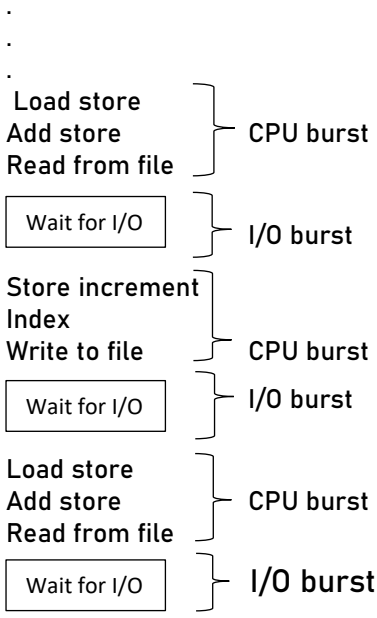


Fig. Alternating sequence of CPU and I/O bursts

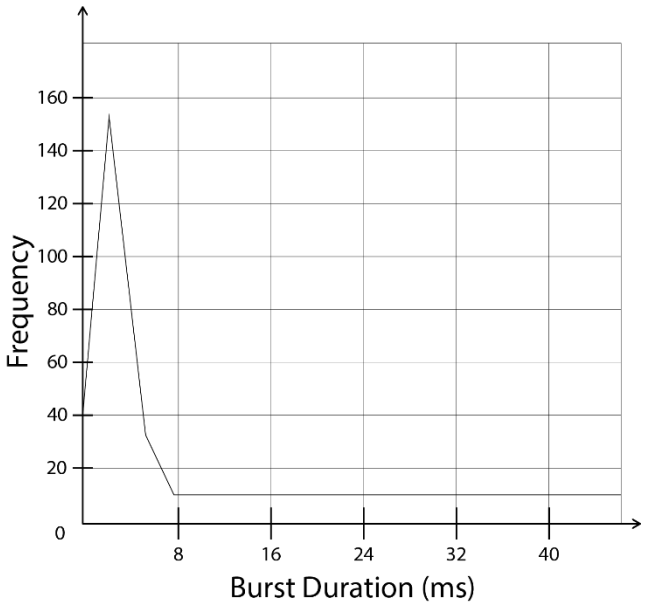


Fig. Histogram of CPU-burst time

- Preemptive scheduling and non-preemptive scheduling –

CPU scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (e.g., I/O request, on invocation of wait for the termination of one of the child processes).
2. When a process switches from the running state to the ready state (e.g., When an interrupt occurs).
3. When a process switches form the waiting state to the ready state (e.g., completion of 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 circumstances 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is non preemptive; otherwise, the scheduling scheme is preemptive.

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until releases the CPU either by terminating or by switching to the waiting state. This scheduling method is only method that can be used on certain hardware platforms, because it does not require the special hardware (e.g., a timer) needed for preemptive scheduling.

In preemptive scheduling the resources (mainly CPU cycles) are allocated to the process for the limited amount of time and then is taken away and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in ready queue till it gets next chance to execute.

In non-preemptive scheduling once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state.
In non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits till the process complete its CPU burst time and then it can allocate the CPU to another process.

Preemptive	Non-Preemptive
In preemptive scheduling the CPU is allocated to the processes for the limited time.	In non-preemptive scheduling the CPU is allocated to the process till it terminated or switches to waiting state.
The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes.	The executing process in non-preemptive scheduling is not interrupted in the middle of execution and wait till its execution.
In preemptive scheduling there is the overhead of switching.	In non-preemptive scheduling has no overhead of switching the process from running state to ready state.
If a high priority process frequently arrives in the ready queue, then the process with low priority has to wait for a long and it may have to starve.	If CPU is allocated to the process having larger burst time, then the processes with small burst time may have to starve.
Flexible	Rigid
Cost associated	No cost associated

- Scheduling criteria –

Many scheduling criteria have been suggested for comparing CPU scheduling algorithms. The characteristics used for comparison can make a substantial difference in the determination of the best algorithm.

- CPU utilization –

We want to keep the CPU as busy as possible. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

- Throughput –

If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes completed per time unit called throughput. For long processes this rate may be 1 process per hour, for short transaction, throughput might be 10 processes per second.

- Turnaround time –

From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of 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.

- Waiting time –

The CPU-scheduling algorithm 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.

- Response time –

In an interactive system, turnaround time may not be the best criterion. Often a process can produce some output fairly early, and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the amount of time it takes t start responding, but not the time that it takes to output that response. The turnaround time is generally limited by the speed of the output device.

- Waiting time – Time difference between turnaround time and burst time.
- Arrival time – Time at which the process arrives in the ready queue.
- Completion time – Time at which process completes its execution.
- Burst time – Time required by a process for CPU execution.
- Turnaround time – Time difference between completion time and arrival time.

❖ Waiting time = Turnaround time – Burst time
❖ Turnaround time = Completion time – Arrival time

➢ Scheduling algorithms –

CPU scheduling deals with the problem of deciding which of the process in the ready queue is to be allocated time CPU.

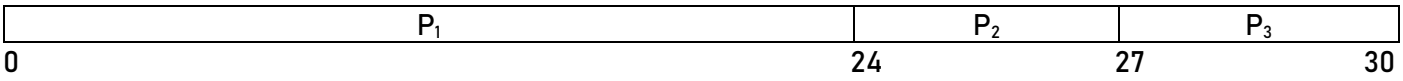
- First-Come, First-Served Scheduling –

  - The simplest CPU scheduling algorithm in the FCFS scheduling algorithm with this scheme the process that requests the CPU first is allocated the CPU first.
  - The implementation of the FCFS policy is easily managed with a FIFO queue.
  - When the CPU is free, it is allocated to the process at the head of the queue.
  - The running process is then removed from the queue.
  - The average waiting time under the FCFS policy, however, is often quite long.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst time given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the process arrives in the order P1, P2, P3 and are reserved in FCFS order, we get the result shown in the following Gantt chart.

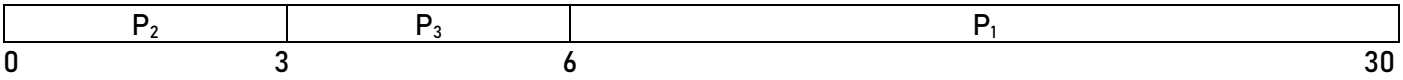


The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2 and 27 milliseconds for process P3.

Thus, the average waiting time is –

$\frac{0+24+27}{3}$ = 17 ms

If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now –

$\frac{6+0+3}{3}$ = 3 ms

This reduction is substantial. Thus, the average waiting time under a FCFS policy is generally not minimal and may vary substantially if the process CPU burst time vary greatly.

- The FCFS scheduling algorithm is non preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
- The FCFS algorithm is particularly troublesome for time-sharing system, where each user needs to get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

  - Advantages –

    - It is the simplest scheduling algorithm.
    - It is easy to implement.

  - Disadvantages –

    - This algorithm is non-preemptive so you have to execute the process fully and after those other processes will be allowed to execute.
    - Throughput is not efficient.
    - FCFS suffers from the convey effect i.e., if a process is having very high burst time and it is coming first, then it will be executed first irrespective of the fact that a process having very low time is there in the ready state.

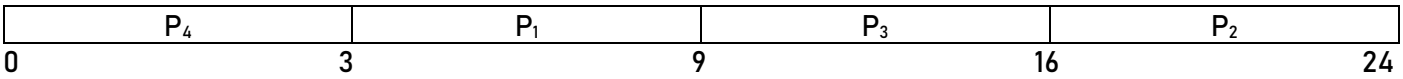
- Shortest Job First Scheduling –

  - This algorithm associates with each process the length of the latter's next CPU burst.
  - When the 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.
  - Note that a more appropriate term would be the shortest next CPU burst, because the scheduling is done by examining the length of the next CPU burst of a process, rather than its total length.

Consider the following set of processes with the length of the CPU burst time given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling we would schedule there processes according to the following Gantt chart:



The waiting time is 3 ms for process P_1, 16 ms for process P_2, 9 ms for process P_3 and 0 ms for process P_4.

Thus, the average waiting time is –

$\frac{3+16+9+0}{4}$ = 7 ms

If we were using the FCFS algorithm then the average waiting time would be 10.25 ms.

- Advantages (non-preemptive) –

  - Short process will be executed first.

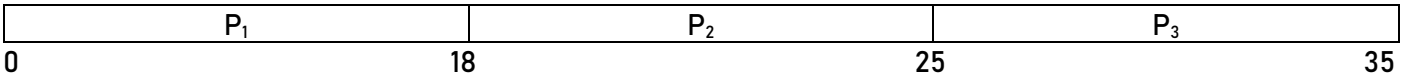
- Disadvantages (non-preemptive) –

  - It may lead to starvation if only short burst time processes are coming in the ready state.

- FCFS –

Process	Arrival Time	Burst Time
P_1	0	18
P_2	2	7
P_3	2	10

Gantt Chart:



Process	Waiting Time (Turnaround Time – Burst Time)	Turnaround Time (Completion Time – Arrival Time)
P_1	0	18
P_2	16	23
P_3	23	33

Total waiting time –

0+16+23 = 39 ms

Average waiting time –

$\frac{39}{3}$ = 13 ms

Total turnaround time –

18+23+33 = 74 ms

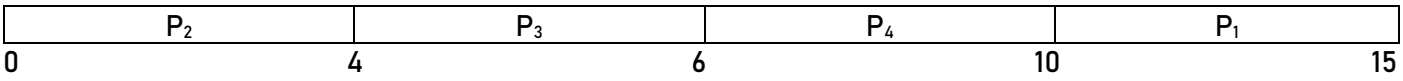
Average turnaround time –

$\frac{74}{3}$ = 24.66 ms

- SJF (non-preemptive) –

Process	Arrival Time	Burst Time
P_1	3	5
P_2	0	4
P_3	4	2
P_4	5	4

Gantt Chart –



Process	Waiting Time (Turnaround Time – Burst Time)	Turnaround Time (Completion Time – Arrival Time)
P_1	7	12
P_2	0	4
P_3	0	2
P_4	1	5

Total waiting time –

7+0+0+1 = 8 ms

Average waiting time –

$\frac{8}{4}$ = 2 ms

Total turnaround time –

12+4+2+5 = 23 ms

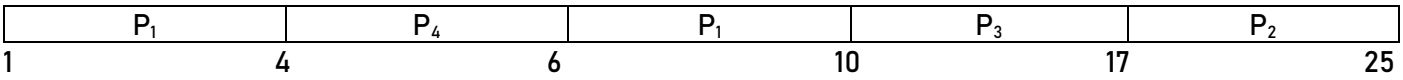
Average turnaround time –

$\frac{23}{4}$ = 5.75 ms

- SJF (preemptive) –

Process	Arrival Time	Burst Time
P_1	1	6
P_2	1	8
P_3	2	7
P_4	3	3

Gantt Chart –



Process	Waiting Time (Turnaround Time – Burst Time)	Turnaround Time (Completion Time – Arrival Time)
P_1	3	9
P_2	16	24
P_3	8	15
P_4	0	3

Total waiting time –

3+16+8+0 = 27 ms

Average waiting time –

$\frac{27}{4}$ = 6.75 ms

Total turnaround time –

9+24+15+3 = 51 ms

Average turnaround time –

$\frac{51}{4}$ = 12.75 ms

- This is the preemptive approach of the shortest job first algorithm.
- Here at every instant of time, the CPU will check for some shortest job. e.g., at time 0 ms we have P_1 as the shortest process. So P_1 will execute for 1 ms and then the CPU will check if some other process is shorter than P_1 or not. If there is no such process, then P_1 will keep on executing for the next 1 ms and if there is some process shorter than P_1 then that process will be executed. This will continue until the process gets executed.
- This algorithm is also known as shortest remaining time first. i.e., we schedule the process based on the shortest remaining time of the processes.

  - Advantages –

    - Short processes will be executed first.

  - Disadvantages –

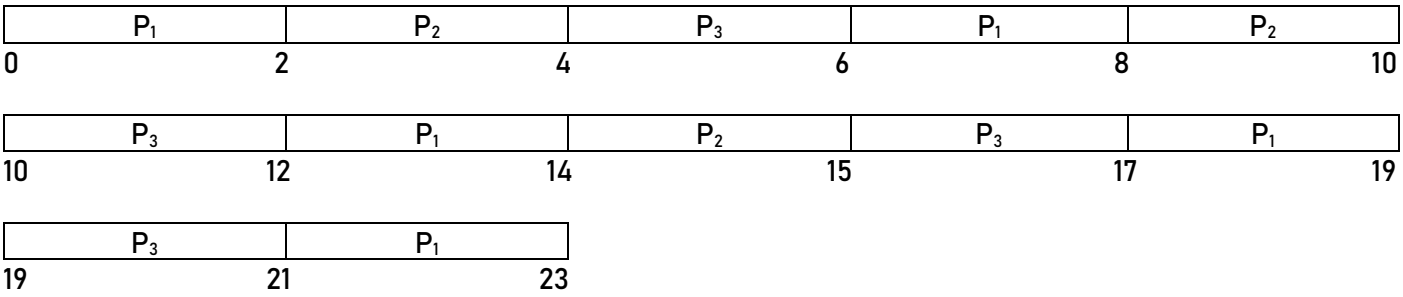
    - It may result in starvation if short processes keep on coming.

- Round-Robin –

  - In this approach of CPU scheduling, we have a fixed time quantum and the CPU will be allocated to a process for that amount of time only.
  - e.g., if we are having three process P_1, P_2 and P_3 and our time quantum is 2 ms, then P_1 will be given 2 ms for its execution, then P_2 will be given 2 ms, then P_3 will be given 2 ms. After one cycle, again P_1 will be given 2 ms, then P_2 will be given 2 ms and so on until the processes complete its execution.
  - It is generally used in the time-sharing environments and there will be no starvation in case of the round-robin.

Process	Arrival Time	Burst Time
P_1	1	6
P_2	1	8
P_3	2	7

Gantt Chart –



Process	Waiting Time (Turnaround Time – Burst Time)	Turnaround Time (Completion Time – Arrival Time)
P_1	13	23
P_2	10	15
P_3	13	21

Total waiting time –

13+10+13 = 36 ms

Average waiting time –

$\frac{36}{3}$ = 12 ms

Total turnaround time –

23+15+21 = 59 ms

Average turnaround time –

$\frac{59}{3}$ = 19.66 ms

  - Advantages –

    - No starvation will be there in round robin because every process will get chance for its execution.
    - Used in time-sharing systems.

  - Disadvantages –

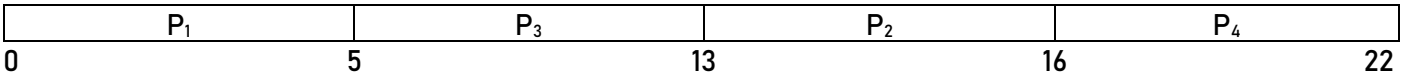
    - We have to perform lots of context switching here, which will keep the CPU idle.

- Priority scheduling (non-preemptive) –

  - In this approach, we have a priority number associated with each process and based on that priority number the CPU selects one process form a list of processes.
  - The priority number can be anything.
  - It is just used to identify which process is having a higher priority and which process is having a lower priority.
  - e.g., you can denote 0 as the highest priority process and 200 as the lowest priority process. Also, the reverse can be true i.e., you can denote 100 as the highest priority and 0 as the lowest priority.

Process	Arrival Time	Burst Time	Priority
P_1	0	5	1
P_2	1	3	2
P_3	2	8	1
P_4	3	6	3

Gantt chart -



Process	Waiting Time (Turnaround Time – Burst Time)	Turnaround Time (Completion Time – Arrival Time)
P_1	0	5
P_2	12	15
P_3	3	11
P_4	13	19

Total waiting time –

0+12+3+13 = 28 ms

Average waiting time –

$\frac{28}{4}$ = 7 ms

Total turnaround time –

5+15+11+19 = 50 ms

Average turnaround time –

$\frac{50}{4}$ = 12.5 ms

  - Advantages –

    - Higher priority processes like system processes are executed first

  - Disadvantages –

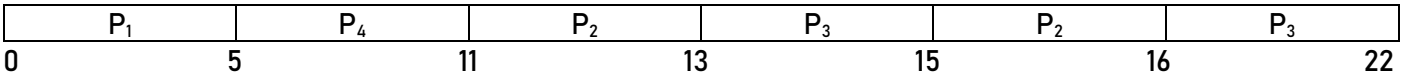
    - It can lead to starvation if only higher priority process comes into the ready state.
    - If the priorities of two processes are the same, then we have to use some other scheduling algorithm.

- Multilevel queue scheduling –

  - In multilevel queue scheduling we divide the whole processes into some batches or queues and each queue is give some priority number.
  - e.g., if there are four processes P_1, P_2, P_3 and P_4 then we can put process P_1 and P_4 in queue 1 and process P_2 and P_3 in queue 2. Now we can assign some priority to each queue, so we can take the queue 1 as having the highest priority and queue 2 as the lowest priority.
  - So, all the processes of the queue 1 will be executed first followed by queue 2. Inside the queue 1 we can apply some other scheduling algorithm for the execution of processes of queue 1. Similar as with the case of queues.
  - So, multiple queues for processes are maintained that are having common characteristics and each queue has its own priority and there is some scheduling algorithm used in each of the queues.

Process	Arrival Time	Burst Time	Queue
P_1	0	5	1
P_2	0	3	2
P_3	0	8	2
P_4	0	6	1

Gantt chart –



- In the above example we have two queues i.e., queue 1 and queue 2. Queue 1 is having higher priority and queue 1 is having FCFS approach and queue 2 is having the round-robin approach (time quantum = 2 ms).
- Since the priority of queue 1 is higher, so queue 1 will be executed first. In the queue 1 we have two processes i.e., P_1 and P_4 and we are using FCFS. So P_1 will be executed followed by P_4. Now the job of the queue is finished. After this the execution of the processes of queue 2 will be started by using the round-robin approach.

Deadlock

- In a multiprogramming system, numerous processes get competed for a finite number of resources. Any process requests resources and as the resources aren't available at that time, the process goes into a waiting state.
- At times, a waiting process is not at all able again to change its state as other waiting processes detain the resources it has requests.
- That condition is termed as deadlock.

- System model –

    - A system model or structure consists of a fixed number of resources to be circulated among some opposing processes.
    - The resources are then partitioned into numerous types, each consisting of some specific quantity of identical instances.
    - Memory space, CPU cycles, directories and files, I/O devices like keyboards, printers and CD-DVD drives are prime examples of resource types.
    - When a system has 2 CPU, then the resource type CPU got two instances.
    - A process must request a resource before using it, and must release the resource after using it.
    - A process may request as many resources as it requires to carry out its designated task.

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:

    1. Request: If the request cannot be granted immediately (e.g., The resources being used by another process) then the requesting process must wait until it can acquire the resource.

    2. Use: The process can operate on the resource (e.g., if the resource is a printer, the process can print on the printer.)

    3. Release: The process releases the resource.

- To illustrate a deadlock state, we consider a system with three types drives.
- Suppose each of three processes holds one of these tape drives. If each process now requests another tape drive, the three processes will be in a deadlock state. Each is waiting for the event "tape drive is released" which can be caused only by one of the other waiting processes.

- Necessary conditions leading to deadlock –

    A deadlock situation can arise if the following four conditions holds simultaneously in a system –

1. Mutual exclusion –

    At least one resource must be held in ta non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait –

    A process must be holding at least one resource and waiting to acquire additional resources that are currently being hold by other processes.

3. No preemption –

    Resources cannot be preempted, that is a resource can be released only voluntarily in the process holding it, after that process has completed its task.

4. Circular wait –

   A set $\{P_0, P_1, \ldots\ldots, P_n\}$ 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 resource that is held by P_n and P_n is waiting for a resource that is held by P_0.

- Deadlock Handling –

1. Deadlock Prevention –

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

Mutual exclusion –

- The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes.
- Sharable resources on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock.
- Read only files are a good example of a sharable resources.
- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- A process never needs to wait for a sharable resource.
- In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition; some resources are intrinsically non-sharable.

Hold and wait –

- To ensure that the hold and wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.
- An alternative protocol allows a process to request resources only when the process has none. A process may request some resources as use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
- To illustrate the difference between their two protocols, we consider a process that copies data from a tape drive to a disk file, sort the disk file and then prints the requests to a printer. If all resources must be requested at the beginning of the process, then the must initially request the tape drive, disk file and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.
- The second method allows the process to request initially only the tape drive and disk file. It copies from the tape drive to the disk, then release both the tape drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

- Disadvantages –

  - Resource utilization may be low
  - Starvation it possible

No preemption –

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it (i.e., the process must wait) then all resources currently being held are preempted. In other words, these resources are implicitly released.

- The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can again 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 form the waiting process and allocate them to the requesting process. If the resources are not either available or held by a waiting process, the requesting process must wait.
- While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restated only when it is allocated the new resources it is requesting and released only resources that were preempted while it was waiting.

Circular wait –

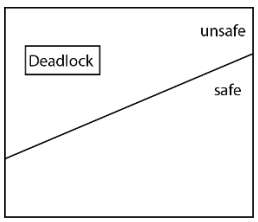
- One way to ensure that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources is an increasing order of enumeration.
- Let $R = \{R_1, R_2, ..., R_m\}$ be the set of resource type. We assign to each resource type a unique 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 $F : R \rightarrow N$, where N is the set of natural numbers.

- We can now consider the following protocol to prevent deadlocks:
    - Each process can request resources 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 $F(R_j) > F(R_i)$.
    - Alternatively, we can require that, whenever a process requests as instance of resource type R_j, it has released any resources R_i such that $F(R_i) \geq F(R_j)$.
    - In other words, in order to request resource R_j a process must first release all R_i such that i >= j.
    - One big challenge in this scheme is determining the relative ordering of the different resources.

- Deadlock avoidance –

Deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that a circular wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

- Safe state –
    - A state is safe of the system can allocate resources to each process (up to its maximum) in order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.
    - A sequence of processes $(P_1, P_2, ..., P_n)$ is a safe sequence for the current allocation state if for each P_i, the resources that P_i, can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j < i$.
    - In this situation, if the resources that process P_i needs are not immediate available, the P_i can wait until all P_j have finished. When they have finished P_i can obtain all of its needed resources, complete its designated task, return its allocated resources and terminated. 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.
- To illustrate, we consider a system with 12 magnetic tape drivers, the P_1 may need as many as 4 and process P_2 may need up to 9 tape drives.
- Suppose that at time to process P_0 is holding 5 tape drivers, process P_1 is holding 2 and process P_2 is holding 2 tape drives.

Process	Maximum needs	Current needs
P_0	10	5
P_1	4	2
P_2	9	2

- At time to, the system is in a safe state. The sequence (P_1, P_0, P_2) satisfies the safety condition.

- Resources allocation graph algorithm –

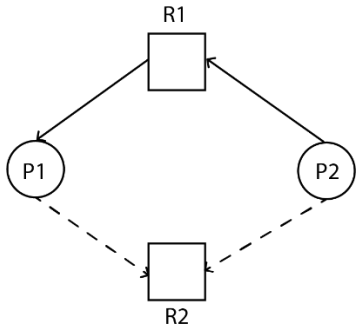
  - Resource allocation graph is explained to us what is the state of the system in term of process and resources.
  - Like how many resources are available, how many are allocated and what is the request of each process.
  - Everything can be represented in terms of the diagram.

  RAG contain vertices and edges. In RAG vertices are two types –

  1. Process vertex – Every process will be represented as a process vertex. Generally, the process vertex will be represented with a circle.
  2. Resource vertex – Every resource will be represented as a resources type.

  There are two types of edges in RAG –

  1. Assignment edge – If you already assign a resource to a process then it is called assign edge.
  2. Request edge – It means in future the process might want some resources to complete the execution.
  3. Claim edge – In addition to the request and assignment edge, we introduce a new type of edge called a claim edge.



  - A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resources 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 request resource R_j the claim edge $P_i \rightarrow R_j$ is converted to a request edge.
  - Similarly, when a resource R_j is released by P_i, the assignment edge $R_j \rightarrow P_i$ is reconverted to claim edge $P_i \rightarrow R_j$.

- Note that the resources must be claimed a priorly in the system. That is before process P_i starts executing all its claimed edges must already appear in the resource allocation graph.
- Suppose that process request resource, the request can be granted only if converting the request edge to an assignment edge does not request in the formation of a cycle in the RAG.
- If no cycle exists then the allocation of the resource will leave the system in a safe state.
- To illustrate this algorithm, we consider the resource allocation graph to figure.
- Suppose that P_2 request R_2, though R_2 is currently free we cannot allocate it to P_2 since this action will create a cycle in the graph.
- A cycle indicates that the system is in an unsafe state of P_1 requests R_2 and P_2 requests R_1 then a deadlock will occur.

o Banker's Algorithm –

- The Banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources.
- Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are 'n' number of account holders in a bank and the total sum of their money is 'S'. If a person applies for a loan, then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned.
- Following data structure are used to implement the Banker's algorithm let 'n' be the number of processes in the system and 'm' be the number of resources types.

▪ Available – A vector of length 'm' indicates the number of available resources of each type. If Available [j] = K, there are K instances of resource types 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 need K more instances of resource type Rj to complete its task. Note that Need [i,j] = Max [i,j] – Allocation [i,j].

- Banker's algorithm consists of safety algorithm and resource request algorithm.

o 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' respectively.
     Initialize Work = available and Finish[i] = safe for i = 1, 2, …, n.

  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;
     Finish[i] = true
     Go to step 2.

  4. If Finish[i] = true for all i, then the system is in safe state. This algorithm may require on order of 'm' x 'n^2' operations to decide whether a state is safe.

- To illustrate this algorithm, we consider a system with five processes P_0 through P_4 and three resource types A, B, C. Resource type A has 7 instances, resource type B has 2 instances and resource type C has 6 instances. Suppose that, at time T_0, we have the following resource allocation state:

Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2	7	4	3
P_1	2	0	0	3	2	2				1	2	2
P_2	3	0	2	9	0	2				6	0	0
P_3	2	1	1	2	2	2				0	1	1
P_4	0	0	2	4	3	3				4	3	1

The content of the matrix Need is defined to Max - Allocation.
Apply the safety algorithm on the given system.

Step 1:

m = 3, n = 5

Work = Available

Work =

3	3	2

Finish =

False	False	False	False	False
0	1	2	3	4

Step 2:

For i = 0

$Need_0$ = 7, 4, 3

Finish[0] is false and $Need_0$ > Work, so P_0 must wait.

Step 2:

For i = 1

$Need_1$ = 1, 2, 2

Finish[1] is false and $Need_1$ < Work, so P_1 must kept in safe sequence.

Step 3:

Work = Work + Allocation

Work =

5	3	2

Finish =

False	Ture	False	False	False
0	1	2	3	4

Step 2:

For i = 2

$Need_2$ = 6, 0, 0

Finish[2] is false and $Need_2$ > Work, so P_2 must wait.

Step 2:

For i = 3

$Need_3$ = 0, 1, 1

Finish[3] is false and $Need_3$ < Work, so P_3 must kept in safe sequence.

Step 3:

Work = Work + Allocation

Work =

7	4	3

Finish =

False	Ture	False	True	False
0	1	2	3	4

Step 2:

For i = 4

$Need_4$ = 4, 3, 1

Finish[4] is false and $Need_4$ < Work, so P_4 must kept in safe sequence.

Step 3:

Work = Work + Allocation

Work =

7	4	5

Finish =

False	Ture	False	True	True
0	1	2	3	4

Step 2:

For i = 0

$Need_0$ = 7, 4, 3

Finish[0] is false and $Need_0$ < Work, so P_0 must kept in safe sequence.

Step 3:

Work = Work + Allocation

Work =

7	5	5

Finish =

True	Ture	False	True	True
0	1	2	3	4

Step 2:

For i = 2

$Need_2$ = 6, 0, 0

Finish[0] is false and $Need_2$ < Work, so P_2 must kept in safe sequence.

Step 3:

Work = Work + Allocation

Work =

10	5	7

Finish =

True	Ture	True	True	True
0	1	2	3	4

Step 4:

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

Hence the system is in safe state.
The safe sequence is P1, P3, P4, P0, P2

- Resource request algorithm –

  - Let $Request_i$ be the request vector for process P_i. If $Request_i[j] = K$, then process P_i wants K instance of resource type R_j. When a request for resources is made by process P_i, then the following actions are taken –

1. If $Request_i < Need_i$, go to Step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If request i ≤ 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 resources allocation state is safe, the transaction is completed and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $Request_i$ and the old resource allocation state is restored.