**Chapter 4**

In Multiprogramming systems, the Operating system schedules the processes on the CPU to have the maximum utilization of it and this procedure is called CPU scheduling. The Operating System uses various scheduling algorithm to schedule the processes.

**Need Scheduling**

In Multiprogramming, if the long term scheduler picks more I/O bound processes then most of the time, the CPU remains idol. The task of Operating system is to optimize the utilization of resources.

If most of the running processes change their state from running to waiting then there may always be a possibility of deadlock in the system. Hence to reduce this overhead, the OS needs to schedule the jobs to get the optimal utilization of CPU and to avoid the possibility to deadlock.

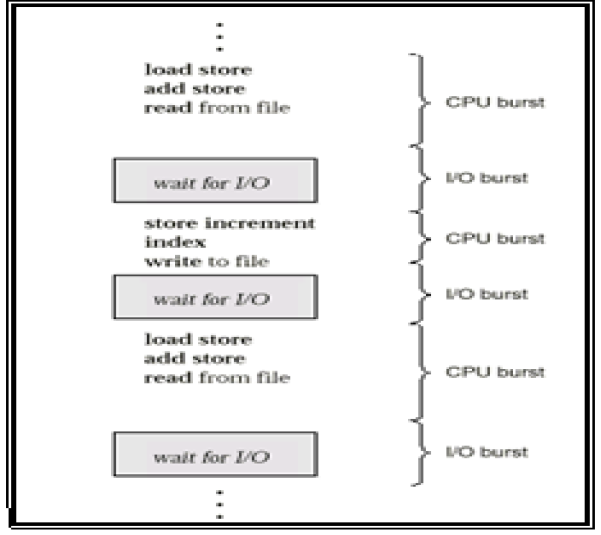
**Explain CPU and I/O burst cycle with the help of diagram.**

CPU burst cycle: - It is a time period when process is busy with CPU.

I/O burst cycle: - It is a time period when process is busy in working with I/O resources.

A process execution consists of a cycle of CPU execution and I/O wait. A process starts its execution when CPU is assigned to it, so process execution begins with a CPU burst cycle. This is followed by an I/O burst cycle when a process is busy doing I/O operations.

A process switch frequently from CPU burst cycle to I/O burst cycle and vice versa. The complete execution of a process starts with CPU burst cycle, followed by I/O burst cycle, then followed by another CPU burst cycle, then followed by another I/O burst cycle and so on. The final CPU burst cycle ends with a system request to terminate execution



**State any four criteria in CPU scheduling**

**CPU utilization**: - In multiprogramming the main objective is to keep CPU as busy as possible.

CPU utilization can range from 0 to 100 percent.

**Throughput**: - It is the number of processes that are completed per unit time.

**Turnaround time**:-The time interval from the time of submission of a process to the time of completion of that process is called as turnaround time.

It is calculated as: Turnaround Time = Waiting Time + Burst Time or End Time – Arrival Time

**Waiting time**: - It is the sum of time periods spent in the ready queue by a process. It is calculated as: Waiting Time = Start Time – Arrival Time

**Response time**:-The time period from the submission of a request until the first response is produced is called as response time.

**Explain the pre-emptive and non-preemptive type of scheduling.**

**Pre-emptive Scheduling**:-Even if CPU is allocated to one process, CPU can be preempted to other process if other process is having higher priority or some other fulfilling criteria.

* Throughput is less
* Only the processes having higher priority are scheduled.
* It doesn’t treat all processes as equal.
* Algorithm design is complex.

Circumstances for preemptive

* Process switch from running to ready state
* Process switch from waiting to ready state

**For e.g.: Round Robin, Priority algorithms**

**Non-Preemptive Scheduling**

Once the CPU has been allocated to a process the process keeps the CPU until it releases CPU either by terminating or by switching to waiting state.

* Throughput is high.
* It is not suitable for RTS.
* Processes having any priority can get scheduled.
* It treats all process as equal.
* Algorithm design is simple.

Circumstances for Non preemptive

* Process switches from running to waiting state
* Process terminates

**For e.g.: FCFS algorithm It is suitable for RTS.**

**Differentiate between pre-emptive and non-pre-emptive scheduling (any 4 points)**

|  |  |
| --- | --- |
| **Pre-emptive Scheduling** | **Non Pre-emptive Scheduling** |
| Even if CPU is allocated to one process, CPU can be preempted to other process if other process is having higher priority or some other fulfilling criteria. | Once the CPU has been allocated to a process the process keeps the CPU until it releases CPU either by terminating or by switching to waiting state. |
| Throughput is less | Throughput is high. |
| Only the processes having higher priority are scheduled. | Processes having any priority can get  scheduled. |
| It doesn’t treat all processes as equal. | It treats all process as equal |
| Algorithm design is complex. | Algorithm design is simple |
| Circumstances for preemptive  (i) Process switch from running to  ready state  (ii) Process switch from waiting to  ready state | Circumstances for Non-preemptive  Process switches from running to  waiting state Process terminates |
| For e.g.: Round Robin, Priority Algorithms | For e.g.: FCFS Algorithm |

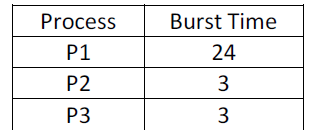
* Completion Time: Time at which process completes its execution.
* Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
* Waiting Time (W.T): Time Difference between turn around time and burst time.
* Waiting Time = Turn Around Time – Burst Time

**Explain first come first served (FCFS) algorithm. Give one example. State any one advantages and one disadvantage.**

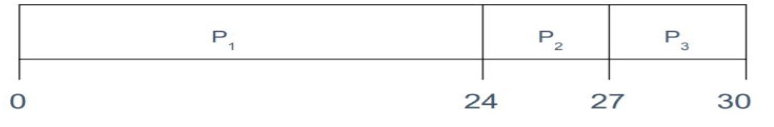
First-Come - First-Served (FCFS) Scheduling FCFS scheduling is non preemptive algorithm.

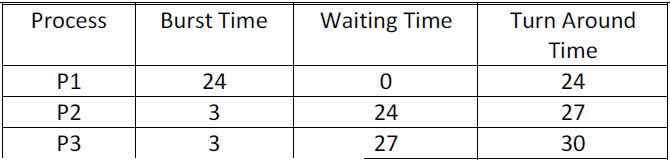
* Once the CPU is allocated to a process, it keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
* In this algorithm, a process, that a request the CPU first, is allocated the CPU first. FCFS scheduling is implemented with a FIFO queue.
* When a process enters the ready queue, its PCB is linked to the tail of the queue.
* When the CPU is available, it is allocated to the process at the head of the queue. Once the CPU is allocated to a process, that process is removed from the queue.
* The process releases the CPU by its own.

Example:



Suppose that the processes arrive in the order: P1, P2, P3 Gantt Chart:





Average waiting time: (0 + 24 + 27)/3 = 17

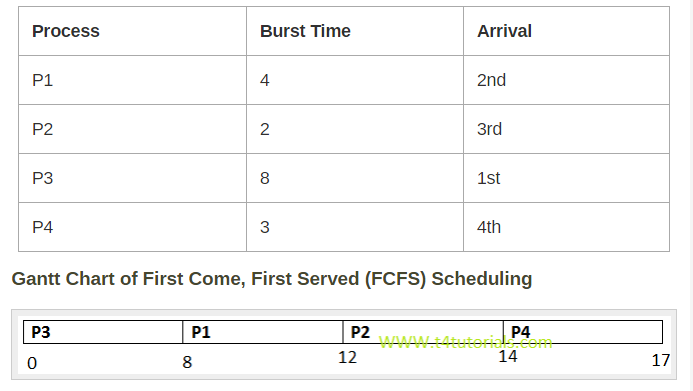
Average turnaround time: (24 + 27 + 30)/3 = 27

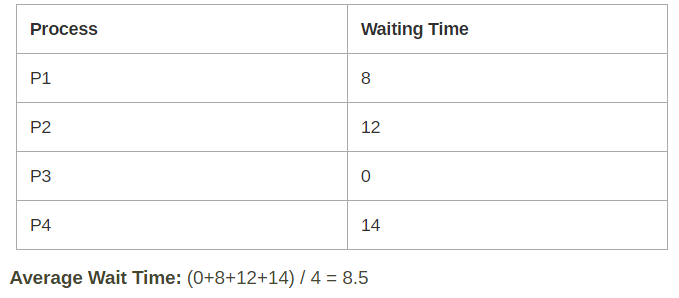
**Advantage:**

It is simple to implement.

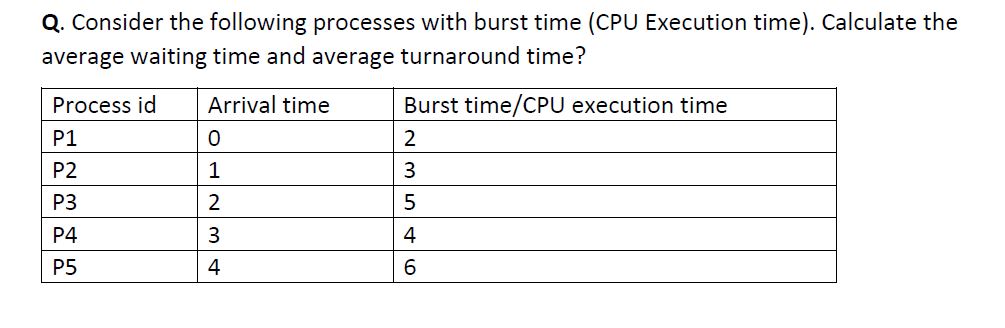
**Disadvantage:**

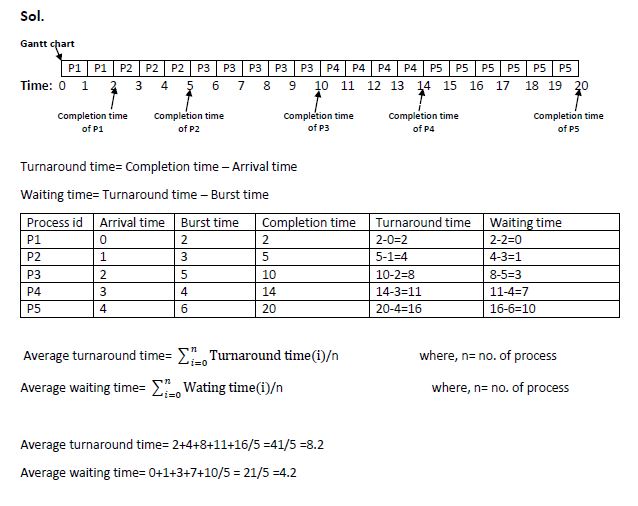
* This scheduling method is non preemptive, that is, the process will run until it finishes. Because of this non preemptive scheduling, short processes which are at the back of the queue have to wait for the long process at the front to finish.
* It is not suitable for real time systems.
* Average waiting time and average turnaround time is more comparatively.











Example: Consider the following table:

|  |  |  |
| --- | --- | --- |
| Process no. | Arrival Time | Burst Time |
| P1 | 0 | 6 |
| P2 | 2 | 1 |
| P3 | 4 | 4 |
| P4 | 5 | 3 |

Find the average waiting time and average turn arround time using FCFS algorithm?

Solution:

 Using FCFS process scheduling algorithm, gantt chart is:

19.jpg

Therefore,

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Waiting Time | Turn Around Time | Completion Time | Process No. | Arrival Time | Burst Time |
| 6 - 6 = 0 | 6 - 0 = 6 | 6 | P1 | 0 | 6 |
| 5 - 1 = 4 | 7 - 2 = 5 | 7 | P2 | 2 | 1 |
| 7 - 4 = 3 | 11 - 4 = 7 | 11 | P3 | 4 | 4 |
| 9 - 3 = 6 | 14 - 5 = 9 | 14 | P4 | 5 | 3 |

 Average Turn Arround Time = (6 + 5 + 7 + 9) / 4 = 6.75

Average Waiting Time = (0 + 4  + 3 + 6) / 4 = 3.25

|  |  |  |
| --- | --- | --- |
| PROCESS | ARRIVAL TIME | BURST TIME |
| P1 | 0 | 24 |
| P2 | 0 | 3 |
| P3 | 0 | 3 |

Gantt chart

|  |  |  |
| --- | --- | --- |
| P1 | P2 | P3 |

0                   24                 27              30

|  |  |  |
| --- | --- | --- |
| PROCESS | WAIT TIME | TURN AROUND TIME |
| P1 | 0 | 24 |
| P2 | 24 | 27 |
| P3 | 27 | 30 |

Total Wait Time

0 + 24 + 27 = 51 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)

51/3 = 17 ms

Total Turn Around Time

24 + 27 + 30 = 81 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)

81 / 3 = 27 ms

In the above example, if order of process arriving is p2, p3, p1 instead of p1, p2, p3 then

Gantt chart

|  |  |  |
| --- | --- | --- |
| P2 | P3 | P1 |

0                     3                     6                  30

|  |  |  |
| --- | --- | --- |
| PROCESS | WAIT TIME | TURN AROUND TIME |
| P1 | 6 | 30 |
| P2 | 0 | 3 |
| P3 | 3 | 6 |

Total Wait Time

6 + 0 + 3 = 9 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)

9/3 = 3 ms

Total Turn Around Time

30 + 3 + 6 = 39 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)

39 / 3 = 13 ms

|  |  |  |
| --- | --- | --- |
| PROCESS | ARRIVAL TIME | BURST TIME |
| P1 | 0 | 80 |
| P2 | 0 | 20 |
| P3 | 0 | 10 |
| P4 | 0 | 20 |
| P5 | 0 | 50 |

Gantt chart

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 |

0                      80                  100           110              130           180

|  |  |  |
| --- | --- | --- |
| PROCESS | WAIT TIME | TURN AROUND TIME |
| P1 | 0 | 80 |
| P2 | 80 | 100 |
| P3 | 100 | 110 |
| P4 | 110 | 130 |
| P5 | 130 | 180 |

Total Wait Time

0 + 80 + 100 + 110 + 130 = 420 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)

420/5 = 84 ms

Total Turn Around Time

80 + 100 + 110 + 130 + 180 = 600 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)

600/5 = 120 ms

|  |  |  |
| --- | --- | --- |
| PROCESS | ARRIVAL TIME | SERVICE TIME |
| P1 | 0 | 8 |
| P2 | 1 | 4 |
| P3 | 2 | 9 |
| P4 | 3 | 5 |

Gantt chart

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P2 | P3 | P4 |

0                             8                     12                 21                   26

|  |  |  |
| --- | --- | --- |
| PROCESS | WAIT TIME | TURN AROUND TIME |
| P1 | 0 | 8 – 0 = 8 |
| P2 | 8 – 1 = 7 | 12 – 1 = 11 |
| P3 | 12 – 2 = 10 | 21 – 2 = 19 |
| P4 | 21 – 3 = 18 | 26 – 3 = 23 |

Total Wait Time

0 + 7 + 10 + 18 = 35 ms

Average Waiting Time = (Total Wait Time) / (Total number of processes)

35/4 = 8.75 ms

Total Turn Around Time

8 + 11 + 19 + 23 = 61 ms

Average Turn Around time = (Total Turn Around Time) / (Total number of processes)

61/4 = 15.25 ms

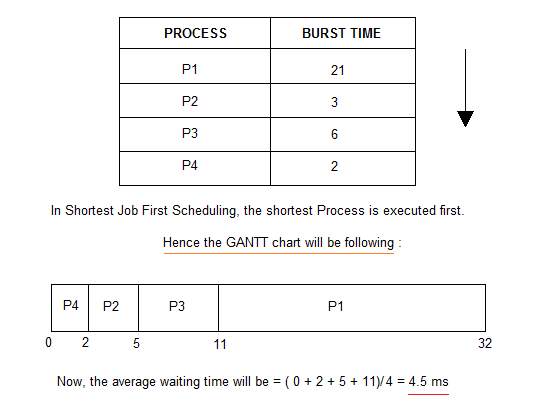
# **Shortest Job First(SJF) Scheduling**

Shortest Job First scheduling works on the process with the shortest **burst time** or **duration** first.

* This is the best approach to minimize waiting time.
* This is used in [Batch Systems](https://www.studytonight.com/operating-system/types-of-os).
* It is of two types:
  1. Non Pre-emptive
  2. Pre-emptive
* To successfully implement it, the burst time/duration time of the processes should be known to the processor in advance, which is practically not feasible all the time.
* This scheduling algorithm is optimal if all the jobs/processes are available at the same time. (either Arrival time is 0 for all, or Arrival time is same for all)

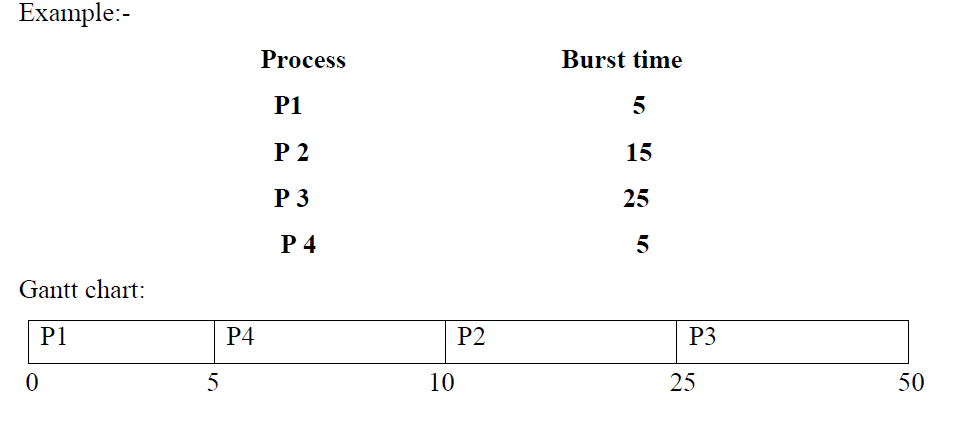
## Non Pre-emptive Shortest Job First

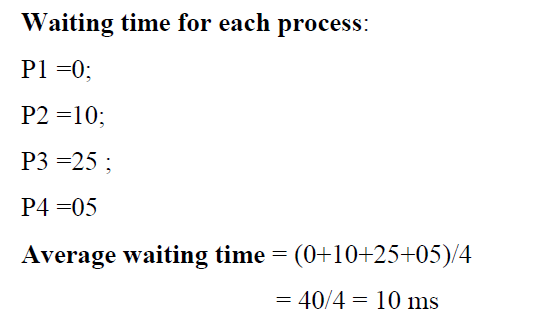
Consider the below processes available in the ready queue for execution, with **arrival time** as 0 for all and given **burst times**.



**Explain SJF algorithm with example. Also calculate average waiting time.**

In this algorithm, each process is associated with the length of the process‟s next CPU burst. When the CPU is available, it is assigned to the process that has smallest next CPU burst. When the next CPU bursts of two processes are same, then FCFS scheduling is used to select one process.





## First Come First Serve (FCFS)

Let's start with the **Advantages:**

* [FCFS algorithm](https://www.studytonight.com/operating-system/first-come-first-serve) doesn't include any complex logic, it just puts the process requests in a queue and executes it one by one.
* Hence, FCFS is pretty simple and easy to implement.
* Eventually, every process will get a chance to run, so starvation doesn't occur.

It's time for the **Disadvantages:**

* There is no option for pre-emption of a process. If a process is started, then CPU executes the process until it ends.
* Because there is no pre-emption, if a process executes for a long time, the processes in the back of the queue will have to wait for a long time before they get a chance to be executed.

**Comparison of Scheduling Algorithms**

## Shortest Job First (SJF)

Starting with the **Advantages:** of [Shortest Job First](https://www.studytonight.com/operating-system/shortest-job-first) scheduling algorithm.

* According to the definition, short processes are executed first and then followed by longer processes.
* The throughput is increased because more processes can be executed in less amount of time.

And the **Disadvantages:**

* The time taken by a process must be known by the CPU beforehand, which is not possible.
* Longer processes will have more waiting time, eventually they'll suffer starvation.

**Note:** Preemptive Shortest Job First scheduling will have the same advantages and disadvantages as those for SJF.

## Round Robin (RR)

Here are some **Advantages:** of using the [Round Robin Scheduling](https://www.studytonight.com/operating-system/round-robin-scheduling):

* Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority.
* Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind.

and here comes the **Disadvantages:**

* The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behavior as FCFS.
* If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.

## Priority based Scheduling

**Advantages** of [Priority Scheduling](https://www.studytonight.com/operating-system/priority-scheduling):

* The priority of a process can be selected based on memory requirement, time requirement or user preference. For example, a high end game will have better graphics, that means the process which updates the screen in a game will have higher priority so as to achieve better graphics performance.

Some **Disadvantages:**

* A second scheduling algorithm is required to schedule the processes which have same priority.
* In preemptive priority scheduling, a higher priority process can execute ahead of an already executing lower priority process. If lower priority process keeps waiting for higher priority processes, starvation occurs.

# **Multilevel Queue Scheduling**

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.

**For example:** A common division is made between foreground(or interactive) processes and background (or batch) processes. These two types of processes have different response-time requirements, and so might have different scheduling needs. In addition, foreground processes may have priority over background processes.

A multi-level 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 the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.

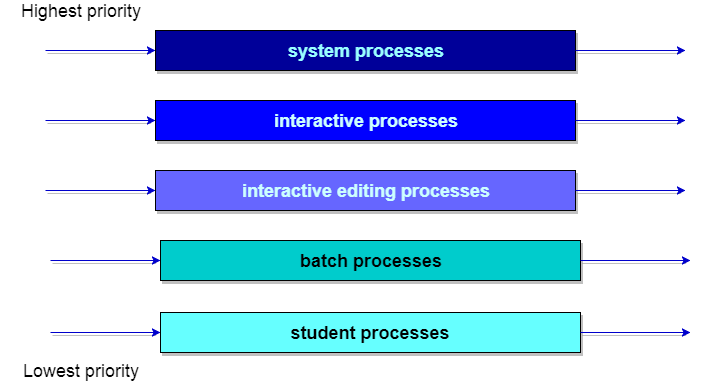
**For example:** separate queues might be used for foreground and background processes. The foreground queue might be scheduled by Round Robin algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. **For example:** The foreground queue may have absolute priority over the background queue.

Let us consider an example of a multilevel queue-scheduling algorithm with five queues:

1. System Processes
2. Interactive Processes
3. Interactive Editing Processes
4. Batch Processes
5. Student Processes

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, 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 will be preempted.



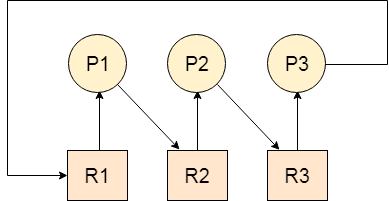
**Define deadlock.**

A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

Let us assume that there are three processes P1, P2 and P3. There are three different resources R1, R2 and R3. R1 is assigned to P1, R2 is assigned to P2 and R3 is assigned to P3.

After some time, P1 demands for R1 which is being used by P2. P1 halts its execution since it can't complete without R2. P2 also demands for R3 which is being used by P3. P2 also stops its execution because it can't continue without R3. P3 also demands for R1 which is being used by P1 therefore P3 also stops its execution.

In this scenario, a cycle is being formed among the three processes. None of the process is progressing and they are all waiting. The computer becomes unresponsive since all the processes got blocked.



**Describe any four condition for deadlock.**

**1. Mutual exclusion:** Only one process at a time can use non-sharable resource.

**2. Hold and wait:** A process is holding at least one resource and is waiting to acquire additional resources held by other processes.

**3. No pre-emption:** A resource can be released only voluntarily by the process holding it after that

process completes its task.

**4. Circular wait:** There exists a set {P0, P1, …, P0} of waiting processes such that P0 is waiting for a

resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a

resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

**Explain in detail how deadlock can be handled.**

There are three different methods for dealing with the deadlock problem:

* We can use a protocol to ensure that the system will never enter a deadlock state
* We can allow the system to enter a deadlock state and then recover.
* We can ignore the problem all together, and pretend that deadlocks never occur in system. This solution is the one used by most operating systems, including UNIX.
  1. To ensure that deadlocks never occur, the system can be 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. These methods prevent deadlocks by constraining how requests for resources can be made.
  2. Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime .with this additional knowledge, we can decide for each request whether or not the process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must be delayed.
  3. 3. If the system does not employ either a deadlock-prevention or deadlock – avoidance algorithm, then a deadlock situation may occur .in this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred, and an algorithm to recover from the deadlock.
  4. 4. If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlock state yet has no way of recognizing what has happened

**List four Deadlock prevention condition and explain the following terms.**

**1) Removal of “No preemption” condition.**

**2) Elimination of “Circular wait” related to deadlock prevention condition.**

**3) Eliminating Mutual Exclusion Condition**

**4) Eliminating Hold and Wait Condition**

**Deadlock prevention conditions:-**

1. Preventing Mutual exclusion condition

2. Preventing Hold and wait condition

3. Preventing No preemption condition

4. Preventing Circular wait condition

**Deadlocks can be prevented by preventing at least one of the four required conditions:**

**Mutual Exclusion**

Shared resources such as read-only files do not lead to deadlocks. Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

**Hold and Wait**

* To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
* Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
* Require that processes holding resources must release them before requesting new esources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
* Either of the methods described above can lead to starvation if a process requires one or more popular resources.

**No Preemption**

* Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
* One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, ( preempted ), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
* Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources and are themselves locked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
* Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

**Circular Wait**

One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing ( or decreasing ) order.

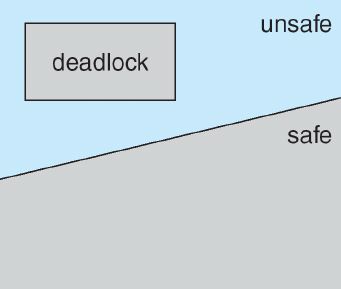
In other words, in order to request resource Rj, a process must first release all Ri such that i >= j. One big challenge in this scheme is determining the relative ordering of the different resources

**Explain Deadlock Avoidance with example.**

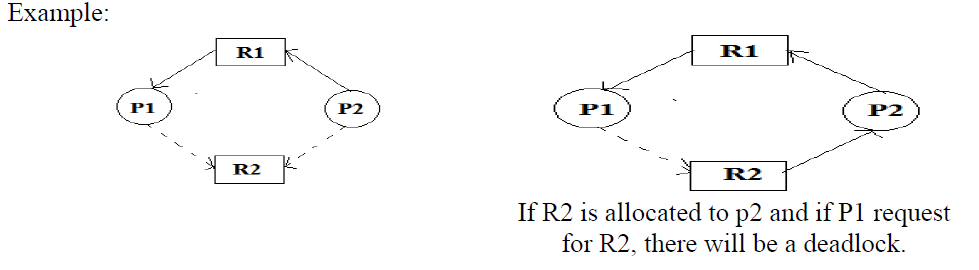
Most prevention algorithms have poor resource utilization, and hence result in reduced throughputs. Instead, we can try to avoid deadlocks by making use prior knowledge about the usage of resources by processes including resources available, resources allocated, future requests and future releases by processes. Most deadlock avoidance algorithms need every process to tell in advance the maximum number of resources of each type that it may need. Based on all this info we may decide if a process should wait for a resource or not and thus avoid chances for circular wait.

**Deadlock can be avoided by following algorithms:**

**Safe State:** If a system is already in a safe state, we can try to stay away from an unsafe state and avoid deadlock. Deadlocks cannot be avoided in an unsafe state. A system can be considered to be in safe state if it is not in a state of deadlock and can allocate resources up to the maximum available. A safe sequence of processes and allocation of resources ensures a safe state. Deadlock avoidance algorithms try not to allocate resources to a process if it will make the system in an unsafe state. Since resource allocation is not done right away in some cases, deadlock avoidance algorithms also suffer from low resource utilization problem.



**Resource Allocation Graph**: A resource allocation graph is generally used to avoid deadlocks. If there are no cycles in the resource allocation graph, then there are no deadlocks. If there are cycles, there may be a deadlock. If there is only one instance of every resource, then a cycle implies a deadlock. Vertices of the resource allocation graph are resources and processes. The resource allocation graph has request edges and assignment edges. An edge from a process to resource is a request edge and an edge from a resource to process is an allocation edge. A calm edge denotes that a request may be made in future and is represented as a dashed line. Based on calm edges we can see if there is a chance for a cycle and then grant requests if the system will again be in a safe state.



**Steps of Banker’s Algorithm:**

This algorithm calculates resources allocated, required and available before allocating resources to any process to avoid deadlock. It contains two matrices on a dynamic basis. Matrix A contains resources allocated to different processes at a given time. Matrix B maintains the resources which are still required by different processes at the same time. Algorithm F: Free resources

**Step 1:** When a process requests for a resource, the OS allocates it on a trial basis.

**Step 2:** After trial allocation, the OS updates all the matrices and vectors. This updating can be done by the OS in a separate work area in the memory.

**Step 3:** It compares F vector with each row of matrix B on a vector to vector basis.

**Step 4:** If F is smaller than each of the row in Matrix B i.e. even if all free resources are allocated to any process in Matrix B and not a single process can complete its task then OS concludes that the system is in unstable state.

**Step 5:** If F is greater than any row for a process in Matrix B the OS allocates all required resources for that process on a trial basis. It assumes that after completion of process, it will release all the recourses allocated to it. These resources can be added to the free vector.

**Step 6:** After execution of a process, it removes the row indicating executed process from both matrices.

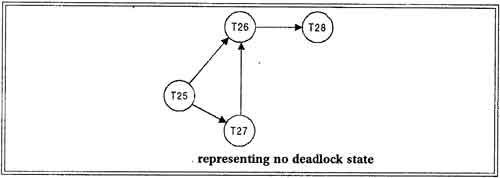
**Step 7:** This algorithm will repeat the procedure step 3 for each process from the matrices and finds that all processes can complete execution without entering unsafe state. For each request for any resource by a process OS goes through all these trials of imaginary allocation and updation. After this if the system remains in the safe state, and then changes can be made in actual matrices.

# **Deadlock Detection**

* A simple way to detect a state of deadlock is with the help of wait-for graph. This graph is constructed and maintained by the system.
* One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction Ti is waiting to lock an item X that is currently locked by a transaction Tj, a directed edge (Ti->Tj).is created in the wait-for graph. When Tj releases the lock(s) on the items that Ti was waiting for, the directed edge is dropped from the wait-for graph.
* We have a state of deadlock if and only if the wait-for graph has a cycle. Then each transaction involved in the cycle is said to be deadlocked.
* To detect deadlocks, the system needs to maintain the wait for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.
* To illustrate these concepts, consider the following wait-for graph in figure. Here: Transaction T25 is waiting for transactions T26 and T27.

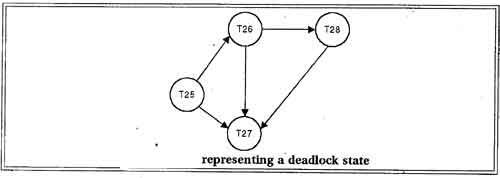
Transactions T27 is waiting for transaction T26.

Transaction T26 is waiting for transaction T28.



This wait-for graph has no cycle, so there is no deadlock state.

Suppose now that transaction T28 is requesting an item held by T27. Then the edge T28 ----------->T27 is added to the wait -for graph, resulting in a new system state as shown in figure.



This time the graph contains the cycle.

T26------>T28------->T27----------->T26

It means that transactions T26, T27 and T28 are all deadlocked.

Invoking the deadlock detection algorithm

The invoking of deadlock detection algorithm depends on two factors:

• How often does a deadlock occur?

• How many transactions will be affected by the deadlock?

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently than usual. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

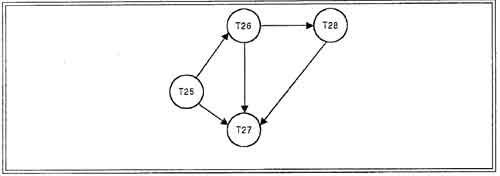
## ****Recovery from Deadlock****

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Choosing which transaction to abort is known as Victim Selection.

## ****Choice of Deadlock victim****

In below wait-for graph transactions T26, T28 and T27 are deadlocked. In order to remove deadlock one of the transaction out of these three transactions must be roll backed.

We should roll back those transactions that will incur the minimum cost. When a deadlock is detected, the choice of which transaction to abort can be made using following criteria:

[](http://ecomputernotes.com/images/Choice%20of%20DeadLock%20victim.jpg)

• The transaction which have the fewest locks

• The transaction that has done the least work

• The transaction that is farthest from completion

## ****Rollback****

Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back. The simplest solution is a total rollback; Abort the transaction and then restart it. However it is more effective to roll back the transaction only as far as necessary to break the deadlock. But this method requires the system to maintain additional information about the state of all the running system.

## ****Problem of Starvation****

In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result this transaction never completes 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.

## ****Detection versus Prevention****

In prevention-based schemes, the abort mechanism is used preemptively in order to avoid deadlocks. On the other hand in detection-based schemes, the transactions in a deadlock cycle hold locks that prevent other transactions from making progress. System put is reduced in case of detection because many transactions may be blocked, waiting to obtain locks currently held by deadlocked transactions.

This is the fundamental trade-off between these prevention and detection approaches to deadlocks: loss of work due to preemptive aborts versus loss of work due to blocked transactions in a deadlock cycle. We can increase the frequency with which we check for deadlock cycles, and thereby reduce the amount of work lost due to blocked transactions, but this entails a corresponding increase in the cost of the deadlock detection mechanism.

Deadlock detection is preferable, if different transactions rarely access the same items t the same time. This can happen if the transactions are short and each transaction locks only *few*items or if the transaction load is light.

On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is quite heavy, it may advantageous to use deadlock prevention scheme.