- **Basic Concepts**

In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. 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 operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

Scheduling of this kind is a fundamental operating-system function. 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 operating-system design.

- **CPU-I/O Burst Cycle**

The success of CPU scheduling depends on an observed property of processes: 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, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution as shown in figure. The durations of CPU bursts have been measured extensively.

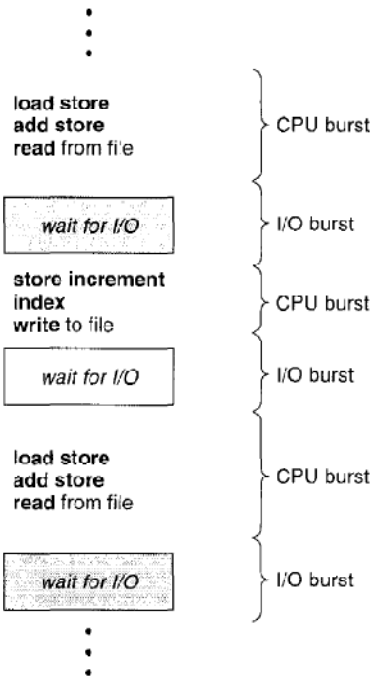


Figure: Alternative sequence of CPU burst and I/O burst

➢ CPU Scheduler

Whenever the CPU becomes idle, the operating system 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 a process from the processes in memory that are ready to execute and allocates the CPU to that process.

The ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

➢ 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 (for example, as the result of an I/O request or an 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 (for example, when an interrupt occurs).

3. When a process switches from the waiting state to the ready state (for example, at completion of I/O).

4. When a process terminates.

For situations 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, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is non-preemptive or cooperative; otherwise, it is preemptive. Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

➢ Dispatcher

Another component involved in the CPU-scheduling flU1ction is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

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

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

• Scheduling Criteria

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

• **CPU utilization:**

We want to keep the CPU as busy as possible. Conceptually, CPU utilization can 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 that are completed per time unit, called *throughput.* For long processes, this rate may be one process per hour; for short transactions, it may 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 a process to the time of completion is the *turnaround time.* Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

• **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 time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, it is desirable to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

- **Scheduling Algorithms**

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU scheduling algorithms.

➢ **First-Come, First-Served Scheduling**

- By far the simplest CPU-scheduling algorithm is the first-come, first-served (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 a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.
- The code for FCFS scheduling is simple to write and understand. The average waiting time under the FCFS policy, however, is often quite long.
- 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.
- There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Advantages-

- It is simple and easy to understand.
- It can be easily implemented using queue data structure.
- It does not lead to starvation.

Disadvantages-

- It does not consider the priority or burst time of the processes.
- It suffers from **convoy effect**.

Example 1

Consider the following example. It consists of four processes, all processes are arrived at the same time and burst time of processes is given in millisecond.

Process	Burst Time
P1	2
P2	3
P3	5
P4	4

1. Draw Gantt Chart.
2. Calculate waiting time (WT) and Turnaround Time (TAT) of each process.
3. Calculate Average waiting time and average turnaround time.

Solution:

1. Gantt Chart

2. Calculation of Turnaround Time (TAT)

Turnaround Time (TAT) = Completion Time – Arrival Time

TAT for process P1 = 2 – 0 = 2 ms

TAT for process P2 = 5 – 0 = 5 ms

TAT for process P3 = 10 – 0 = 10 ms

TAT for process P4 = 14 – 0 = 14 ms

Calculation of Waiting Time (WT)

Waiting Time (WT) = Turnaround Time – Burst Time

WT for process P1 = 2 – 2 = 0 ms

WT for process P2 = 5 – 3 = 2 ms

WT for process P3 = 10 – 5 = 5 ms

WT for process P4 = 14 – 4 = 10 ms

3. Calculation of Average Waiting Time and Average Turnaround Time

Average Waiting Time = (0 + 2 + 5 + 10) / 4 = 17 / 4 = 4.25 ms

Average Turnaround Time = (2 + 5 + 10 + 14) / 4 = 31 / 4 = 7.75 ms.

Example 2

Consider an example. It consists of four processes. Their arrival time and burst time as shown below.

Process	Arrival Time	Burst Time
P1	**0**	**2**
P2	**1**	**3**
P3	**2**	**4**
P4	**3**	**5**

1. **Draw Gantt chart.**
2. **Calculate waiting time and turnaround time for each process.**
3. **Calculate average waiting time and average turnaround time.**

Solution:

1. **Gantt Chart:**

2. **Calculation of Turnaround Time and Waiting Time**

   Turnaround Time (TAT) = Completion Time – Arrival Time

   TAT for process P1 = 2 – 0 = 2 units.

   TAT for process P2 = 5 – 1 = 4 units.

   TAT for process P3 = 9 – 2 = 7 units.

   TAT for process P4 = 14 – 3 = 11 units.

   Waiting Time (WT) = Turnaround Time – Burst Time

   WT for process P1 = 2 – 2 = 0 units

   WT for process P2 = 4 – 3 = 1 units

   WT for process P3 = 9 – 2 = 7 units.

   WT for process P4 = 14 – 3 = 11 units

3. **Calculation of Average Turnaround Time and Average waiting Time**

   Average TAT = (2 + 4 + 7 + 11) / 4 = 24 / 4 = 6 units

   Average WT = (0 +1 + 7 + 11) / 4 = 19 / 4 = 4.75 units.

- **Shortest Job First (SJF):**

- A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm.
- This algorithm associates with each process the length of the process's next CPU burst.
- When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- Note that a more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
- The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes.
- Moving a short process before a long one decrease the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.
- The SJF algorithm can be either preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing.
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.
- Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first scheduling**.

Example:

Consider the following set of processes where the length of CPU burst is given in millisecond. Solve using SJF non-preemptive CPU Scheduling algorithm.

Processes: P1, P2, P3, P4

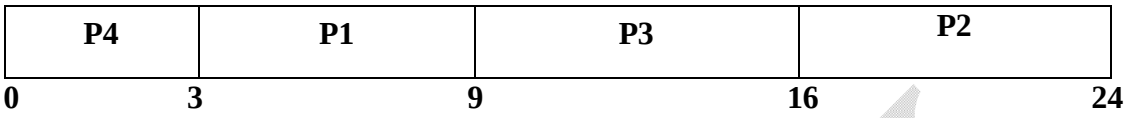
Burst Time: 6, 8, 7, 3.

1) **Draw Gantt chart.**
2) **Calculate Turnaround Time and Waiting Time for each Process.**

3) **Calculate average Turnaround Time and average waiting Time.**

Solution:

1) **Gantt Chart:**



2) **Calculation of Turnaround Time:**

Turnaround Time (TAT) = Completion Time – Arrival Time

TAT for process P1 = 9 – 0 = 9 ms.

TAT for process P2 = 24 – 0 = 24 ms.

TAT for process P3 = 16 – 0 = 16 ms.

TAT for process P4 = 3 – 0 = 3 ms.

Waiting Time (WT) = Turnaround Time – Burst Time

WT for process P1 = 9 – 6 = 3 ms

WT for process P2 = 24 – 8 = 16 ms

WT for process P3 = 16 – 7 = 9 ms.

WT for process P4 = 3 – 3 = 0 ms

3) **Calculation of Average Turnaround Time and Average waiting Time**

Average TAT = (9 + 24 + 16 + 3) / 4 = 52 / 4 = 13 ms

Average WT = (3 +16 + 9 + 0) / 4 = 28 / 4 = 7 ms.

Example:

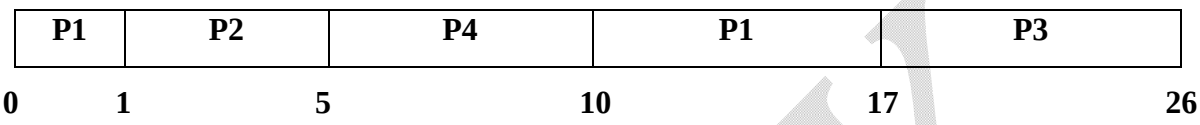
Consider the following four processes, with the length of the CPU burst given in milliseconds. Solve using SJF preemptive scheduling algorithm.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

1) **Draw Gantt chart.**
2) **Calculate Turnaround Time and Waiting Time for each Process.**
3) **Calculate average Turnaround Time and average waiting Time.**

Solution:

1) **Gantt Chart:**



2) **Calculation of Turnaround Time:**

Turnaround Time (TAT) = Completion Time – Arrival Time

TAT for process P1 = 17 – 0 = 17 ms.

TAT for process P2 = 5 – 1 = 4 ms.

TAT for process P3 = 26 – 2 = 24 ms.

TAT for process P4 = 10 – 3 = 7 ms.

Waiting Time (WT) = Turnaround Time – Burst Time

WT for process P1 = 17 – 8 = 9 ms

WT for process P2 = 4 – 4 = 0 ms

WT for process P3 = 24 – 9 = 15 ms.

WT for process P4 = 7 – 5 = 2 ms

3) **Calculation of Average Turnaround Time and Average waiting Time**

Average TAT = (17 + 4 + 24 + 7) / 4 = 52 / 4 = 13 ms

Average WT = (9 +0 + 15 + 2) / 4 = 26 / 4 = 6.5 ms.

- **Priority Scheduling**

- The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

- We discuss scheduling in terms of high priority and low priority. Some systems use low numbers to represent low priority; others use low numbers for high priority.
- This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

Example:

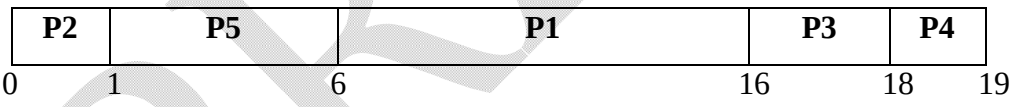
Consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2…, P5 with the length of the CPU burst given in milliseconds:

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

1) Draw Gantt chart.
2) Calculate turnaround time and waiting time for each process.
3) Calculate average turnaround time and average waiting time.

Solution:

1) Gantt Chart



To calculate turnaround time, waiting time, average turnaround time and average waiting time, follow above procedure.

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

- A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked.
- A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.
- Generally, one of two things will happen. Either the process will eventually be run or the computer system will eventually crash and lose all unfinished low-priority processes.
- A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

➢ **Round Robin (RR) Scheduling**

- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily.
- The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system.
- A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.
- The average waiting time under the RR policy is often long.

Advantages-

- It gives the best performance in terms of average response time.
- It is best suited for time sharing system, client server architecture and interactive system.

Disadvantages-

- It leads to starvation for processes with larger burst time as they have to repeat the cycle many times.
- Its performance heavily depends on time quantum.
- Priorities cannot be set for the processes.

Example:

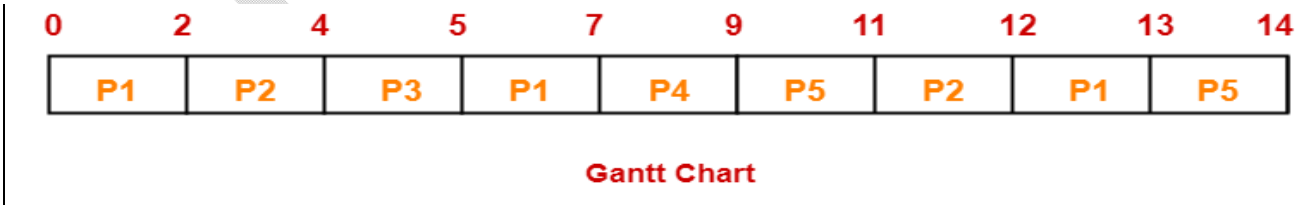
Consider the set of 5 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time and average turnaround time.

Solution:

Gantt Chart-



Gantt Chart

- Turn Around time = Completion time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Arrival time	Burst time	Completion Time	Turn Around time	Waiting time
P1	0	5	13	13 – 0 = 13	13 – 5 = 8
P2	1	3	12	12 – 1 = 11	11 – 3 = 8
P3	2	1	5	5 – 2 = 3	3 – 1 = 2
P4	3	2	9	9 – 3 = 6	6 – 2 = 4
P5	4	3	14	14 – 4 = 10	10 – 3 = 7

Now,

- Average Turn Around time = (13 + 11 + 3 + 6 + 10) / 5 = 43 / 5 = 8.6 unit
- Average waiting time = (8 + 8 + 2 + 4 + 7) / 5 = 29 / 5 = 5.8 unit

• Multiple Processor Scheduling

In multiple-processor scheduling multiple CPU's are available and hence Load Sharing becomes possible. However multiple processor scheduling is more complex as compared to single processor scheduling. In multiple processor scheduling there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality; we can use any processor available to run any process in the queue.

➢ Approaches to Multiple-Processor Scheduling –

One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors executes only the **user code**. This is simple and reduces the need of data sharing. This entire scenario is called **Asymmetric Multiprocessing**.

A second approach uses **Symmetric Multiprocessing** where each processor is **self scheduling**. All processes may be in a common ready queue or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

➢ Processor Affinity –

Processor Affinity means a processes has an **affinity** for the processor on which it is currently running.

When a process runs on a specific processor there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor and as a result successive

memory accesses by the processes are often satisfied in the cache memory. Now if the process migrates to another processor, the contents of the cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most of the SMP (symmetric multiprocessing) systems try to avoid migration of processes from one processor to another and try to keep a process running on the same processor. This is known as **Processor Affinity**.

There are two types of processor affinity:

1. **Soft Affinity –** When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
2. **Hard Affinity –** Some systems such as Linux also provide some system calls that support Hard Affinity which allows a process to migrate between processors.

- **Load Balancing –**

Load Balancing is the phenomenon which keeps the workload evenly distributed across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of process which are eligible to execute. On SMP(symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.

There are two general approaches to load balancing:

1. **Push Migration –** In push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the processes from overloaded to idle or less busy processors.
2. **Pull Migration –** Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

- **Multicore Processors –**

In multicore processors, **multiple processor** cores are places on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a

separate physical processor. **SMP systems** that use multicore processors are faster and consume **less power** than systems in which each processor has its own physical chip.

However multicore processors may **complicate** the scheduling problems. When processor accesses memory then it spends a significant amount of time waiting for the data to become available. This situation is called **Memory Stall**. It occurs for various reasons such as cache miss, which is accessing the data that is not in the cache memory. In such cases the processor can spend upto fifty percent of its time waiting for data to become available from the memory. To solve this problem recent hardware designs have implemented multithreaded processor cores in which two or more hardware threads are assigned to each core. Therefore if one thread stalls while waiting for the memory, core can switch to another thread.

There are two ways to multithread a processor:

1. **Coarse-Grained Multithreading –** In coarse grained multithreading a thread executes on a processor until a long latency event such as a memory stall occurs, because of the delay caused by the long latency event, the processor must switch to another thread to begin execution. The cost of switching between threads is high as the instruction pipeline must be terminated before the other thread can begin execution on the processor core. Once this new thread begins execution it begins filling the pipeline with its instructions.
2. **Fine-Grained Multithreading –** this multithreading switch between threads at a much finer level mainly at the boundary of an instruction cycle. The architectural design of fine grained systems includes logic for thread switching and as a result the cost of switching between threads is small.

- **Virtualization and Threading –**

In this type of **multiple-processor** scheduling even a single CPU system acts like a multiple-processor system. In a system with Virtualization, the virtualization presents one or more virtual CPU's to each of virtual machines running on the system and then schedules the use of physical CPU'S among the virtual machines. Most virtualized environments have one host operating system and many guest operating systems. The host operating system creates and manages the virtual machines and each virtual machine has a guest operating system installed and applications running

within that guest. Each guest operating system may be assigned for specific use cases, applications, and users, including time sharing or even real-time operation. Any guest operating-system scheduling algorithm that assumes a certain amount of progress in a given amount of time will be negatively impacted by the virtualization.

Virtualizations can thus undo the good scheduling-algorithm efforts of the operating systems within virtual machines

- ## Real time systems

Real time system means that the system is subjected to real time, i.e., response should be guaranteed within a specified timing constraint or system should meet the specified deadline. For example: flight control system, real time monitors etc.

Types of real time systems based on timing constraints:

1. **Hard real time system –** This type of system can never miss its deadline. Missing the deadline may have disastrous consequences. The usefulness of result produced by a hard real time system decreases abruptly and may become negative if tardiness increases. Tardiness means how late a real time system completes its task with respect to its deadline. Example: Flight controller system.
2. **Soft real time system –** This type of system can miss its deadline occasionally with some acceptably low probability. Missing the deadline have no disastrous consequences. The usefulness of result produced by a soft real time system decreases gradually with increase in tardiness. Example: Telephone switches.

Reference model of real time system: Our reference model is characterized by three elements:

1. **A workload model:** It specifies the application supported by system.
2. **A resource model:** It specifies the resources available to the application.
3. **Algorithms:** It specifies how the application system will use resources.

Terms related to real time system:

- **Job –** A job is a small piece of work that can be assigned to a processor and may or may not require resources.

- **Task –** A set of related jobs that jointly provide some system functionality.
- **Release time of a job –** It is the time at which job becomes ready for execution.
- **Execution time of a job –** It is the time taken by job to finish its execution.
- **Deadline of a job –** It is the time by which a job should finish its execution. Deadline is of two types: absolute deadline and relative deadline.
- **Response time of a job –** It is the length of time from release time of a job to the instant when it finishes.
- Maximum allowable response time of a job is called its relative deadline.
- Absolute deadline of a job is equal to its relative deadline plus its release time.
- Processors are also known as active resources. They are essential for execution of a job. A job must have one or more processors in order to execute and proceed towards completion. Example: computer, transmission links.
- Resources are also known as passive resources. A job may or may not require a resource during its execution. Example: memory, mutex.
- Two resources are identical if they can be used interchangeably else they are heterogeneous.