OPERATING SYSTEM

UNIT-2 – Chapter – 5

# 1. CPU SCHEDULING:

- CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold (in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU.
- The aim of CPU scheduling is to make the system efficient, fast and fair.
- 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 from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

The idea of multiprogramming is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer, the CPU would then sit idle; 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 operating system takes the CPU away from that process and given the CPU to another process.

Scheduling is a fundamental operating-system function.

## 1. 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 (Figure).
- The durations of CPU bursts have been measured extensively.
- Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure. The curve is generally characterized as exponential or hyper exponential, with a large number of short CPU bursts and a small number of long CPU bursts.

- An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

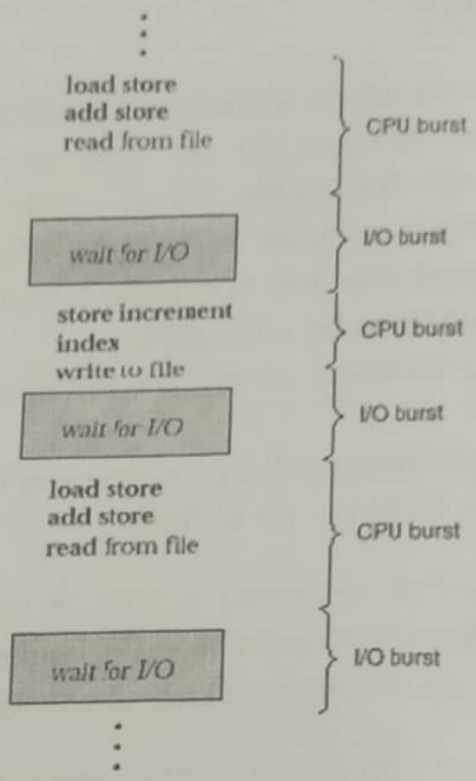



**Figure : Alternating sequence of CPU and I/O bursts.**

## 2. 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.
- Note that 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.

## 3. 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**

- 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**.

## Non-Preemptive Scheduling

- 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.

- This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems.
- It is the only method that can be used on certain hardware platforms, because It does not require the special hardware (for example: a timer) needed for preemptive scheduling.

**Preemptive Scheduling**

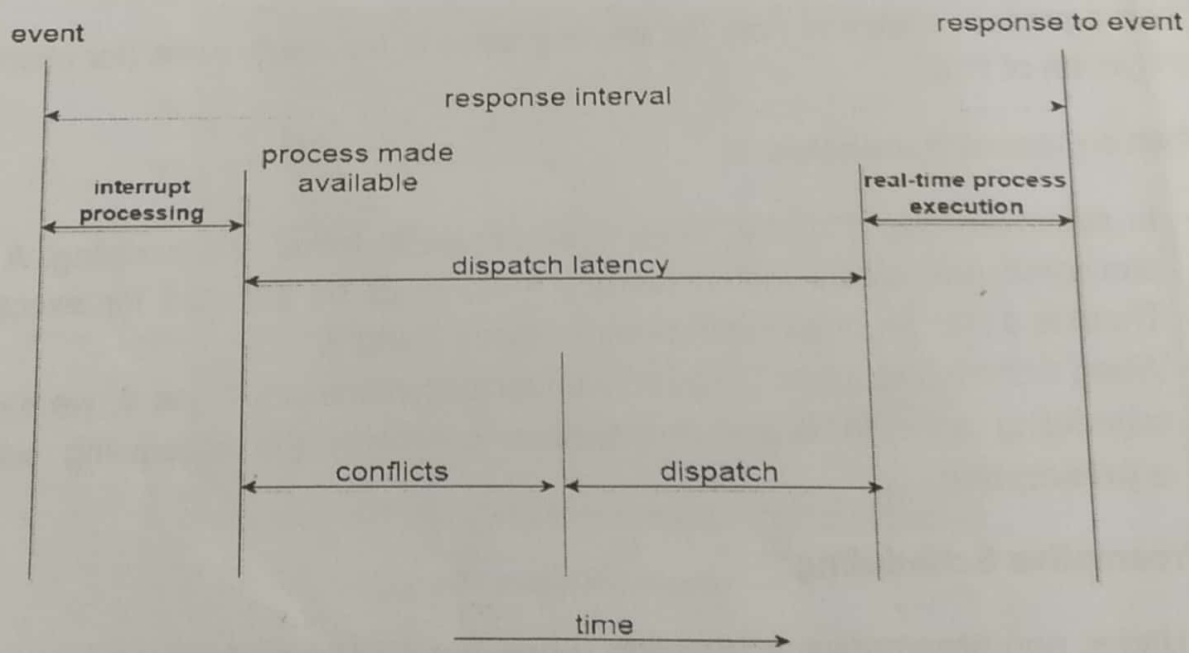
In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

## 4. Dispatcher:

- Another component involved in the CPU scheduling function is the **dispatcher.**
- The dispatcher is the module that given control of the CPU to the process selected by the **short-term scheduler**.
    - Switching context
    - Switching to user mode
    - Jumping to the proper location in the user program to restart that program

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

Dispatch Latency can be explained using the below figure:



## 2. Scheduling Criteria:

There are many different criteria to check when considering the **"best"** scheduling algorithm, they are:

1. **CPU Utilization**
2. **Throughput**
3. **Turnaround Time**
4. **Waiting Time**
5. **Response Time**

**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:** 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 Turn around time, waiting time, and response time.
- For example, to guarantee that all users get good service, we may want to For
- example, to guarantee that all users get good service, we may want to minimize
- the maximum response time.minimize the maximum response time.

To decide which process to execute first and which process to execute last to achieve maximum CPU utilization, computer scientists have defined some algorithms, and they are:

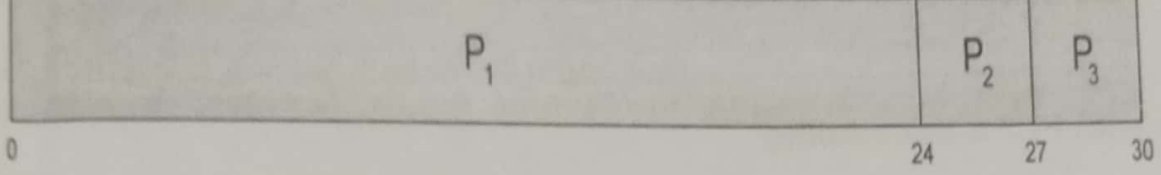
1. **First Come First Serve(FCFS) Scheduling**
2. **Shortest-Job-First(SJF) Scheduling**
3. **Priority Scheduling**
4. **Round Robin(RR) Scheduling**
5. **Multilevel Queue Scheduling**
6. **Multilevel Feedback Queue Scheduling**

## 1. First Come First Serve 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.
- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| PI | 24 |
| P2 | 3 |
| P3 | 3 |

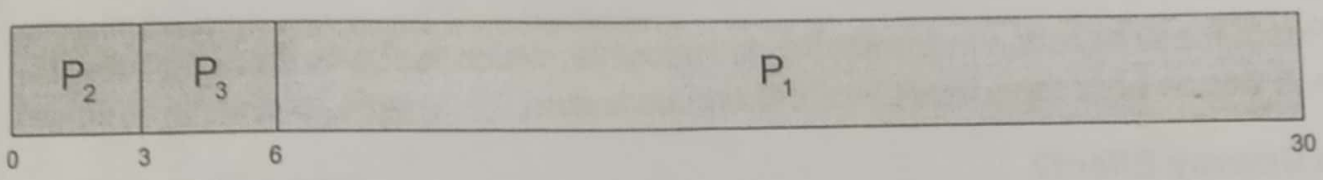
- If the processes arrive in the order PI, P2, P3, and are served in FCFS order, we get the result shown in the following **Gantt chart:**




- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

- The waiting time is 0 milliseconds for process PI, 24 milliseconds for process P2, and 27 milliseconds for process P3.
- Thus, the average waiting time is (0+ 24 + 27)/3 = 17 milliseconds.
- If the processes arrive in the order P2, *P3*, P1,however, the results will be as shown in the following Gantt chart:
- Suppose that the processes arrive in the order:

  $P_2$ , $P_3$ , $P_1$
- The Gantt chart for the schedule is:




- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$

- The average waiting time is now (6 + 0 + 3)/3 =3 milliseconds. This reduction is substantial.
- Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.
- In the "First come first serve" scheduling algorithm, as the name suggests, the process which arrives first, gets executed first, or we can say that the process which requests the CPU first, gets the CPU allocated first.

- First Come First Serve, is just like **FIFO**(First in First out) Queue data structure, where the data element which is added to the queue first, is the one who leaves the queue first.

- This is used in Batch Systems.

- It's **easy to understand and implement** programmatically, using a Queue data structure, where a new process enters through the **tail** of the queue, and the scheduler selects process from the **head** of the queue.

- A perfect real life example of FCFS scheduling is **buying tickets at ticket counter**.

**Problems with FCFS Scheduling**

Below we have a few shortcomings or problems with the FCFS scheduling algorithm:

1. It is **Non Pre-emptive** algorithm, which means the **process priority** doesn't matter.

If a process with very least priority is being executed, more like **daily routine backup** process, which takes more time, and all of a sudden some other high priority process arrives, like **interrupt to avoid system crash**, the high priority process will have to wait, and hence in this case, the system will crash, just because of improper process scheduling.

2. Not optimal Average Waiting Time.
3. Resources utilization in parallel is not possible, which leads to **Convoy Effect**, and hence poor resource(CPU, I/O etc) utilization.

**What is Convoy Effect?**

- Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time.
- This essentially leads to poort utilization of resources and hence poor performance.

- 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 1/0.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems where it is important that each user 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.

**Program for FCFS Scheduling:**

- Here we have a simple C++ program for processes with **arrival time** as 0.
- In the program, we will be calculating the **Average waiting time** and **Average turn around time** for a given array of **Burst times** for the list of processes.

```
/* Simple C++ program for implementation
of FCFS scheduling */
```

(9)

```
#include<iostream>

using namespace std;

// function to find the waiting time for all processes
void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
   // waiting time for first process will be 0
   wt[0] = 0;

   // calculating waiting time
   for (int i = 1; i < n ; i++)
   {
      wt[i] =  bt[i-1] + wt[i-1];
   }
}

// function to calculate turn around time
void findTurnAroundTime( int processes[], int n, int bt[], int wt[], int tat[])
{
   // calculating turnaround time by adding
   // bt[i] + wt[i]
   for (int i = 0; i < n ; i++)
   {
      tat[i] = bt[i] + wt[i];
   }
}

// function to calculate average time
void findAverageTime( int processes[], int n, int bt[])
{
   int wt[n], tat[n], total_wt = 0, total_tat = 0;

   // function to find waiting time of all processes
   findWaitingTime(processes, n, bt, wt);

   // function to find turn around time for all processes
   findTurnAroundTime(processes, n, bt, wt, tat);

   // display processes along with all details
   cout << "Processes  "<< " Burst time  "<< " Waiting time  " << " Turn around
time\n";

   // calculate total waiting time and total turn around time
   for (int i = 0; i < n; i++)
   {
      total_wt = total_wt + wt[i];
      total_tat = total_tat + tat[i];
      cout << "   " << i+1 << "\t\t" << bt[i] <<"\t    "<< wt[i] <<"\t\t  " << tat[i] <<endl;
   }
```

```
    cout << "Average waiting time = "<< (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "<< (float)total_tat / (float)n;
}

// main function
int main()
{
    // process ids
    int processes[] = { 1, 2, 3, 4};
    int n = sizeof processes / sizeof processes[0];

    // burst time of all processes
    int  burst_time[] = {21, 3, 6, 2};

    findAverageTime(processes, n,  burst_time);

    return 0;
}
```

```
Processes Burst time Waiting time Turn around time
1 21 0 21
2 3 21 24
3 6 24 30
4 2 30 32


Average waiting time = 18.75
Average turn around time = 26.75
```

Here we have simple formulae for calculating various times for given processes:

**Completion Time**: Time taken for the execution to complete, starting from arrival time.

**Turn Around Time**: Time taken to complete after arrival. In simple words, it is the difference between the Completion time and the Arrival time.

**Waiting Time**: Total time the process has to wait before it's execution begins. It is the difference between the Turn Around time and the Burst time of the process.

For the program above, we have considered the **arrival time** to be 0 for all the processes, try to implement a program with variable arrival times.

**First Come First Serve (FCFS)**

**Advantages:**

- FCFS algorithm 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.

**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.

## 2. Shortest Job First(SJF) Scheduling:

- 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 two processes have the same length next CPU bursts, 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.
- We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

As an **example of SJF scheduling**, consider the following set of processes, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

0 3 9 16 24

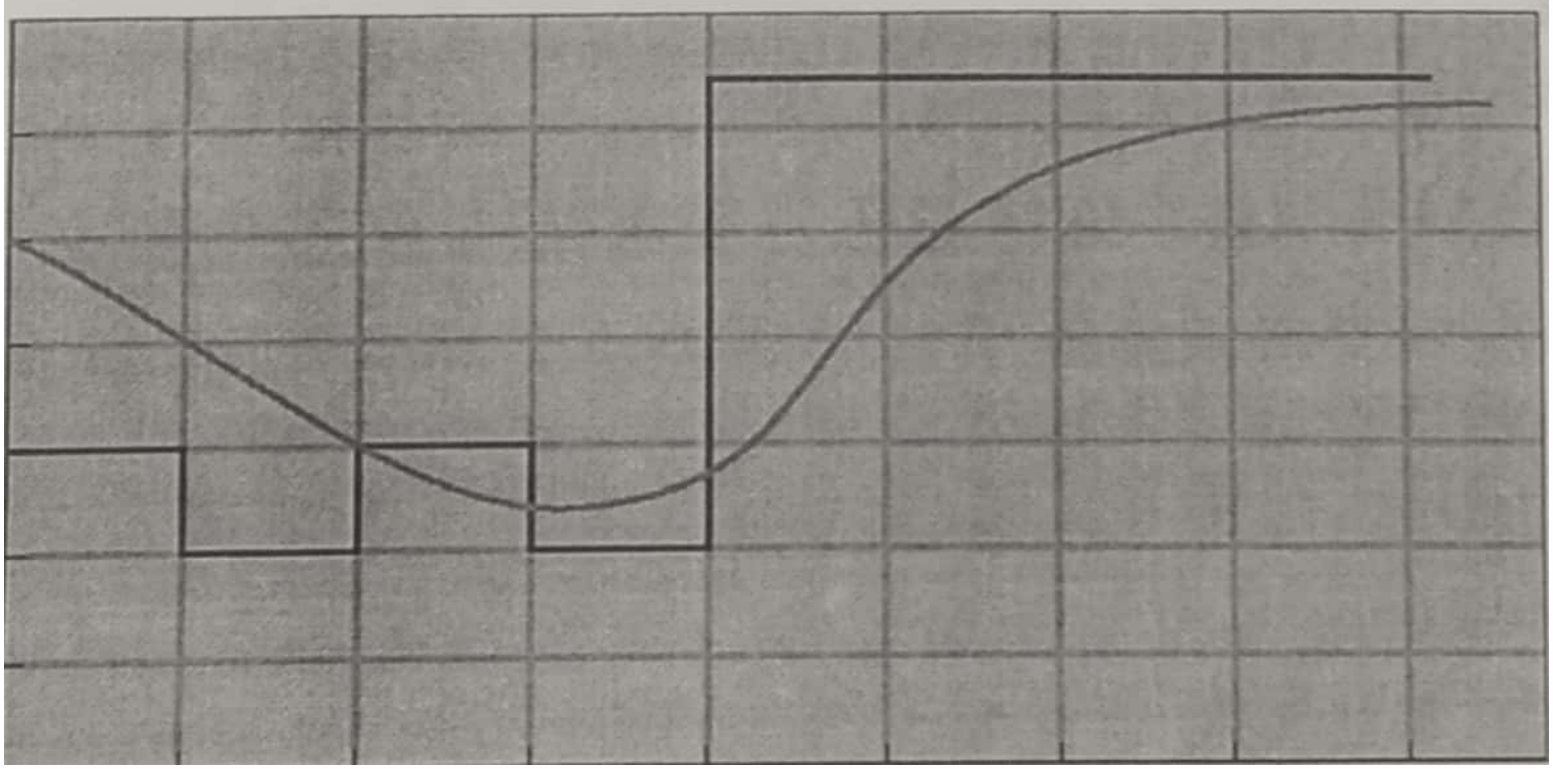
Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

- Using SJF scheduling, we would schedule these processes according to the following Gantt chart:
- The waiting time is 3 milliseconds for process PI, 16 milliseconds for process *P2*, 9 milliseconds for process *P3*, and 0 milliseconds for process P4 .
- Thus, the average waiting time is (3 + 16 + 9 + 0)/4= 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.
- 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 decreases 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 till executing.
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.

...the currently executing process,

1 5 10 17

rage waiting time = [(10-1)+(1-1)+(17-2)+(5-3))]/4 = 26/4 = 6.5 r
ess P1 is started at time 0, since it is the only process in
ess P2 arrives at time 1.
remaining time for process *P1* (7 milliseconds) is larger tha
ired by process *P2* (4 milliseconds), so process *P1* is pree
ess *P2* is scheduled.
average waiting time for this example is ((10 - 1) + (1 1) +
)/4 = 26/4 = 6.5 milliseconds.
preemptive SJF scheduling would result in an average wai
milliseconds.
Can only estimate the length
Can be done by using the length of previous CPU bursts, using
veraging



- $\tau_{n+1} = \tau_n$
- Recent history does not count.
- $\alpha = 1$
- $\tau_{n+1} = \alpha\, t_n$
- Only the actual last CPU burst counts.
- If we expand the formula, we get:
- $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \ldots + (1 - \alpha)^j \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0$
- Since both $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

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

Starting with the **Advantages:** of 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

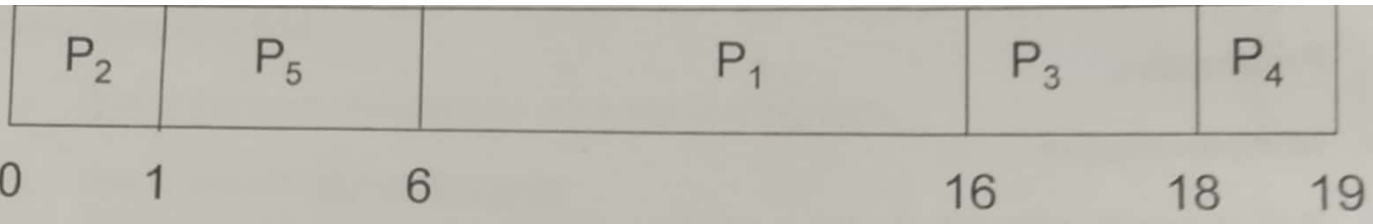
## 3. Priority Scheduling

- Priority is assigned for each process
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the **highest priority** (smallest integer ≡ highest priority)
- **Preemptive**
- **Nonpreemptive**
- SJF is priority scheduling where priority is the next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

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.
- An SJF algorithm is simply a priority algorithm where the priority *(p)* is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
- Note that we discuss scheduling in terms of ***high*** **priority** and ***low*** **priority.**
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority.
- Some systems use low numbers to represent low priority; others use low numbers for high priority. As an 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 |
|---|---|---|




Average waiting time = 8.2 msec

**Priority based Scheduling**

**Advantages** of Priority Scheduling:

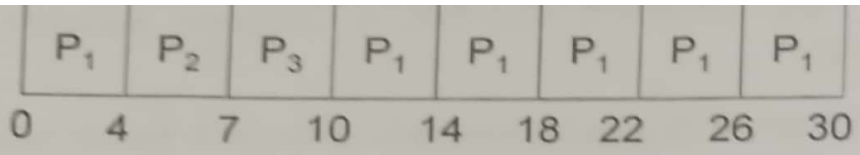
- The priority of a process can be ...

## 4. Round Robin Scheduling:

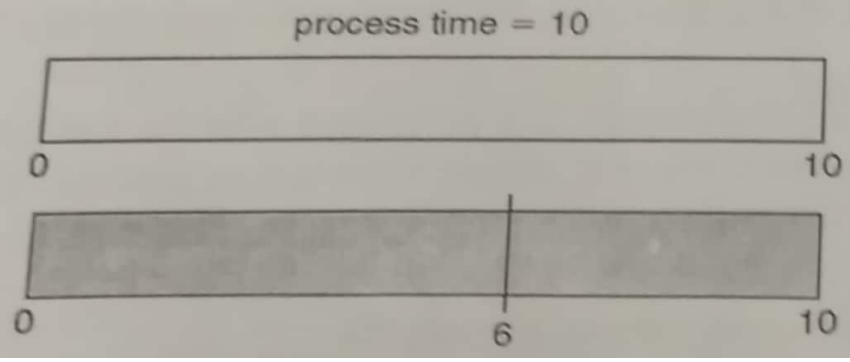
- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- A small unit of time, called a time quantum or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds.
- 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.
- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preemptied and other process executes for given time period.
- Context switching is used to save states of preemptied processes.
- The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

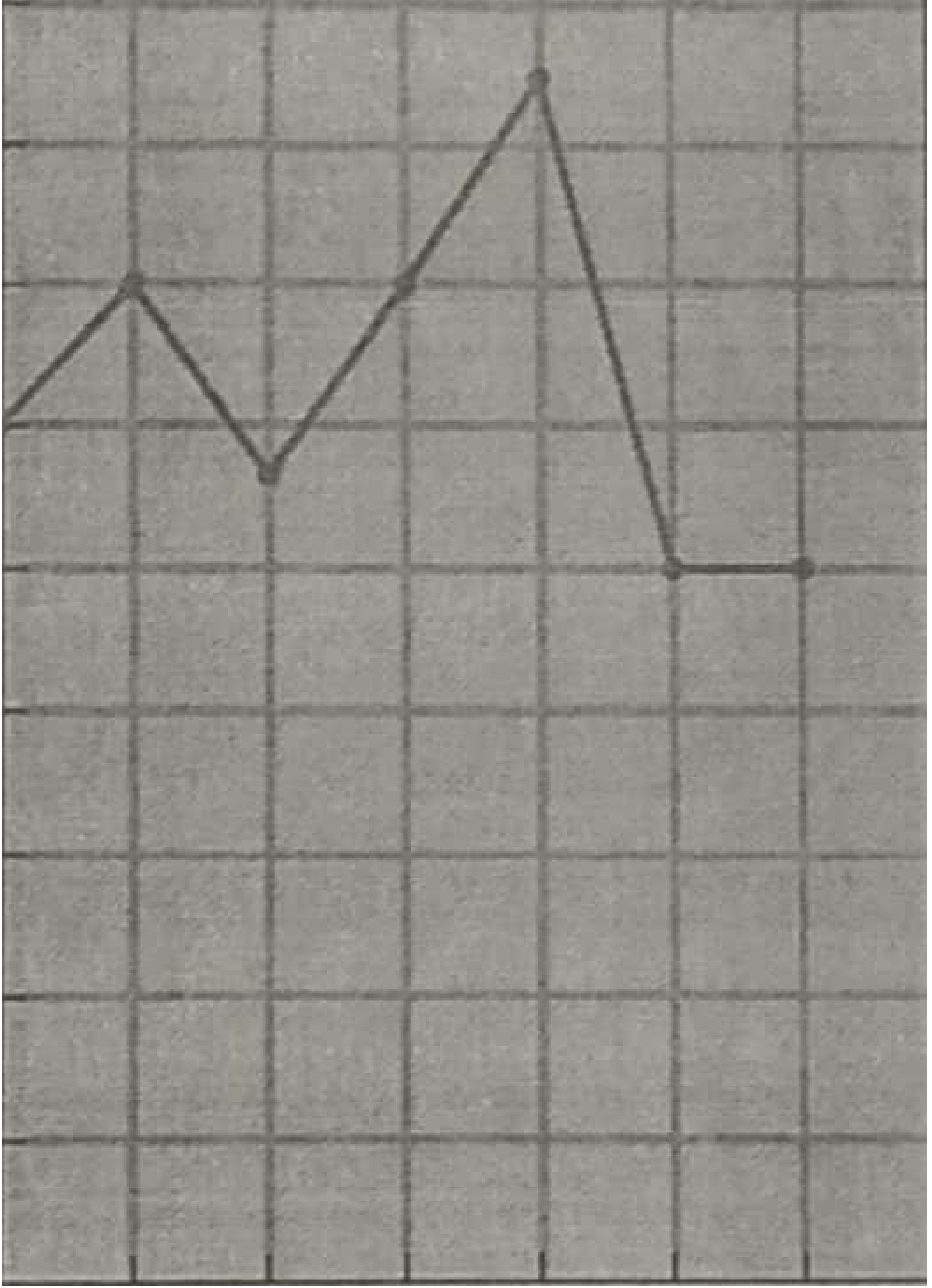
| Process | Burst Time |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |

- If we use a time quantum of 4 milliseconds, then process PI gets the first 4 milliseconds.
- Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2.
- Since process P2 does not need 4 milliseconds, it quits before its time quantum expires.
- The CPU is then given to the next process, process *P3*.
- Once each process has received 1 time quantum, the CPU is returned to process PI for an additional time quantum. The resulting RR schedule is




- The **average waiting time is 17/3 = 5.66 milliseconds.**
- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row.
- If a 5.3 Scheduling Algorithms 165 process's CPU burst exceeds 1 time quantum, that process is **preempted** and is put back in the ready queue.
- The RR scheduling algorithm is thus preemptive.
- If there are 11 processes in the ready queue and the time quantum is *q*, then each process gets **1/n** of the CPU time in chunks of at most *q* time units.
- Each process must wait no longer than **(n - 1) x q** time units until its next time quantum.
- **For example**, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.





**Round Robin (RR)**

**Advantages:** of using the 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.

**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.

## 4. 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.

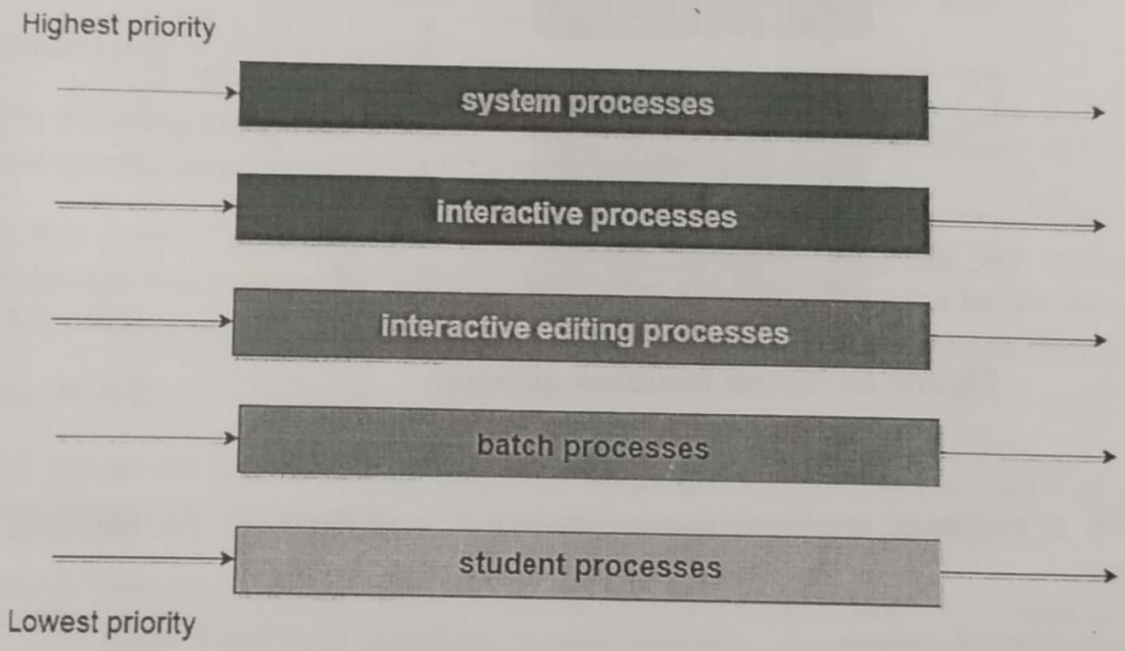
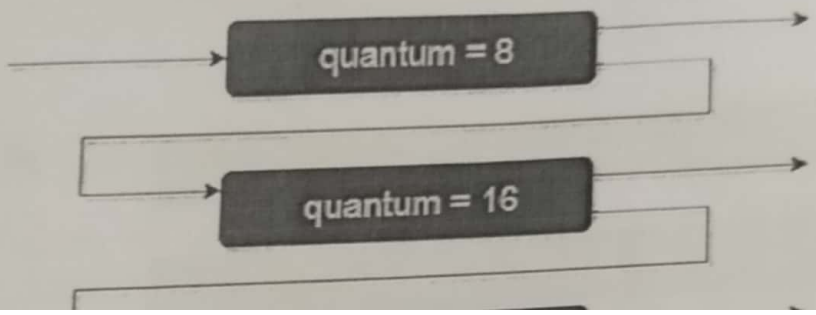



**Figure: Multilevel queue scheduling.**

- Another possibility is to time-slice among the queues.
- Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.

- In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system.
- Processes do not move between queues.
- This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.
- Multilevel feedback queue scheduling, however, allows a process to move between queues.
- The idea is to separate processes with different CPU-burst characteristics.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- This form of aging prevents starvation.



- The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm.
- It can be configured to match a specific system under design.
- Unfortunately, it also requires some means of selecting values for all the parameters to define the best scheduler.
- Although a multilevel feedback queue is the **most general scheme**, it is also the **most complex.**

### Comparison of Scheduling Algorithms

By now, you must have understood how CPU can apply different scheduling algorithms to schedule processes. Now, let us examine the advantages and disadvantages of each scheduling algorithms that we have studied so far.

### Usage of Scheduling Algorithms in Different Situations

Every scheduling algorithm has a type of a situation where it is the best choice. Let's look at different such situations:

**Situation 1:**

The incoming processes are short and there is no need for the processes to execute in a specific order.

In this case, FCFS works best when compared to SJF and RR because the processes are short which means that no process will wait for a longer time. When each process is executed one by one, every process will be executed eventually.

**Situation 2:**

The processes are a mix of long and short processes and the task will only be completed if all the processes are executed successfully in a given time.

Round Robin scheduling works efficiently here because it does not cause starvation and also gives equal time quantum for each process.

**Situation 3:**

The processes are a mix of user based and kernel based processes.

Priority based scheduling works efficiently in this case because generally kernel based processes have higher priority when compared to user based processes.

- If multiple CPUs are available, load sharing among them becomes possible.
- The scheduling problem becomes more complex.
- We concentrate in this discussion on systems in which the processors are identical (homogeneous) in terms of their functionality.
- We can use any available processor to run any process in the queue.

## 1.Approaches to Multiple-Processor Scheduling:

- One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor-the master server.
- The other processors execute only user code.

- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
- Two approaches: **Asymmetric** processing and **symmetric** processing.

- **Asymmetric multiprocessing**
- **Symmetric multiprocessing (SMP)** Processor affinity – process has affinity for processor on which it is currently running
- **soft affinity**
- **hard affinity**

**Asymmetric multiprocessing (ASMP)**

- One processor handles all scheduling decisions, I/O processing, and other system activities
- The other processors execute only user code
- Because only one processor accesses the system data structures, the need for data sharing is reduced

**Symmetric multiprocessing (SMP)**

- Each processor schedules itself
- All processes may be in a common ready queue or each processor may have its own ready queue
- Either way, each processor examines the ready queue and selects a process to execute
- Efficient use of the CPUs requires load balancing to keep the workload evenly distributed
- In a **Push** migration approach, a specific task regularly checks the processor loads and redistributes the waiting processes as needed
- In a **Pull** migration approach, an idle processor pulls a waiting job from the queue of a busy processor

- Virtually all modern operating systems support SMP, including Windows XP, Solaris, Linux, and Mac OS X.

### 2. Processor Affinity:

- Processor affinity – process has affinity for processor on which it is currently running
- **soft affinity**
- **hard affinity**

- The high cost of invalidating and re-populating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor.

- This is known as processor affinity, meaning that a process has an affinity for the processor on which it is currently running.

- Processor affinity takes several forms.

- When an operating system has a policy of attempting to keep a process running on the same processor-but not guaranteeing that it will do so- we have a situation known as **soft affinity**.

- Here, it is possible for a process to migrate between processors. Some systems -such as Linux-also provide system calls that support **hard affinity**, thereby allowing a process to specify that it is not to migrate to other processors.

## 5. Real-Time CPU Scheduling:

**Real-time system:**

- "A real-time system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computation, but also on the physical instant at which these results are produced".
- "A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment".
- Real-time computing is divided into two types.
- **Hard real time**
- **Soft real time**

- **Hard real time:** Hard real time systems are required to complete a critical task within a guaranteed amount of time. Generally, a process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O.

- The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as **resource reservation.**

- **Soft real-time:** Soft real-time computing is less restrictive. It requires that critical processes receive priority over less fortunate ones. Although adding soft real-time functionality to a time-sharing system may cause an unfair allocation of resoures and unfor

Another component involved in the CPU scheduling function 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:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program from where it left last time.

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time taken by the dispatcher to stop one process and start another process is known as the **Dispatch Latency**. Dispatch Latency can be explained using the below figure:

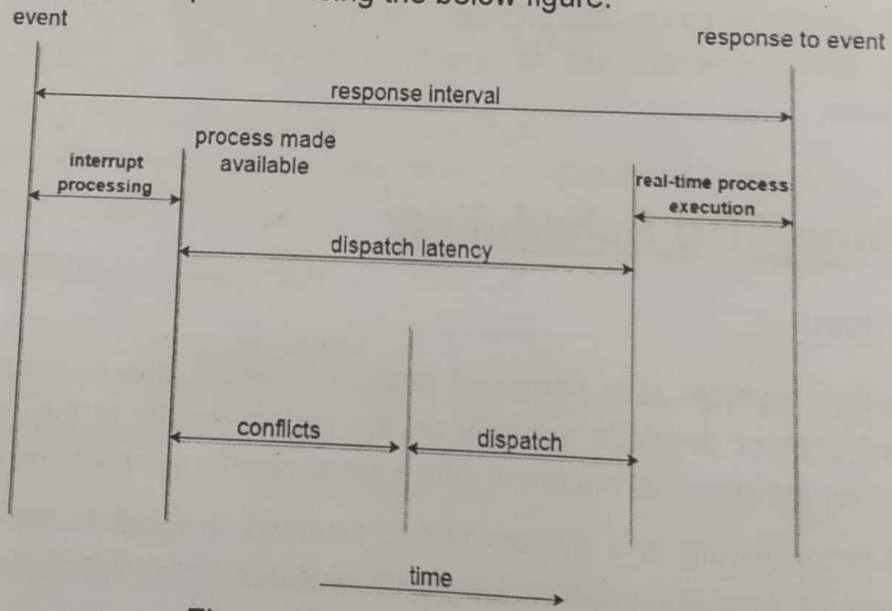



Figure: Dispatch Latency