Unit 2CLASS 3

What is **Process Scheduling**?

The act of determining which process is in the ready state, and should be moved to the running state is known as Process Scheduling.

The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

Scheduling fell into one of the two general categories:

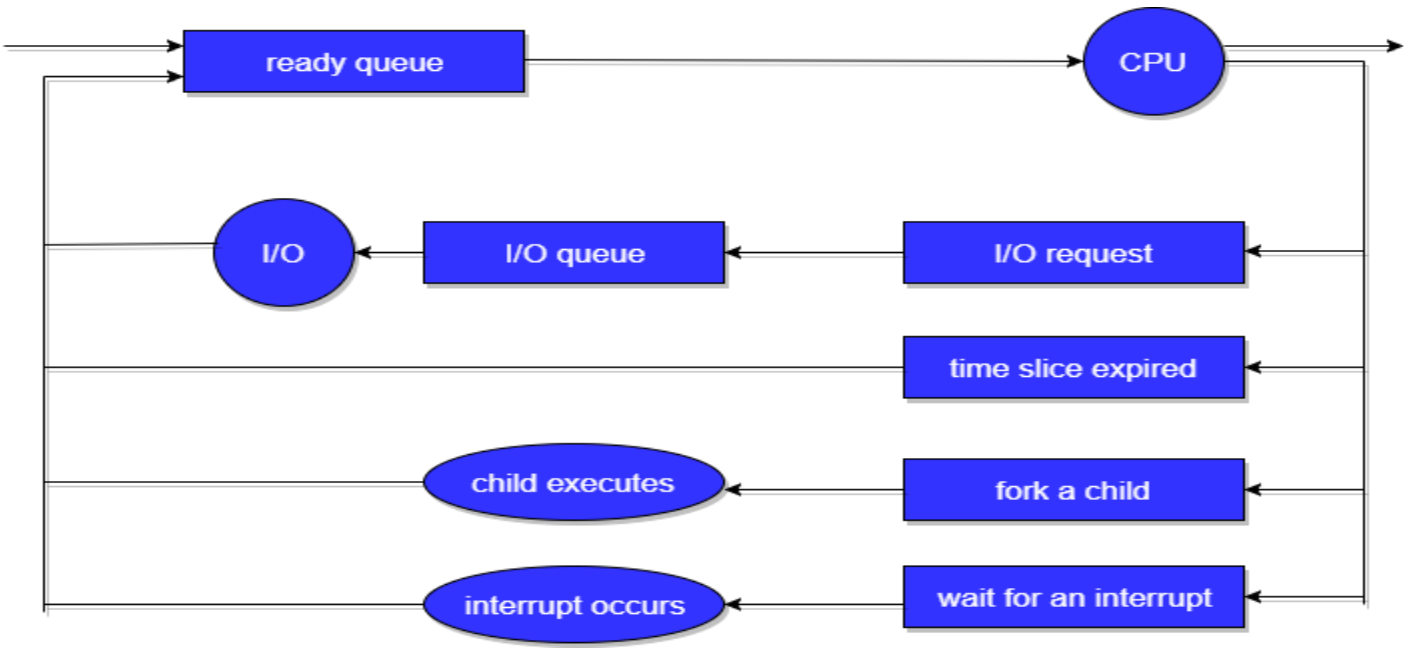
- **Non-Preemptive Scheduling:** When the currently executing process gives up the CPU voluntarily.
- **Pre-emptive Scheduling:** When the operating system decides to favor another process, pre-empting the currently executing process.

What are Scheduling Queues?

- All processes, upon entering into the system, are stored in the Job Queue.
- Processes in the Ready state are placed in the Ready Queue.
- Processes waiting for a device to become available are placed in Device Queues. There are unique device queues available for each I/O device.

A new process is initially put in the Ready queue. It waits in the ready queue until it is selected for execution(or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

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



In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Types of Schedulers

There are three types of schedulers available:

1. Long Term Scheduler
2. Short Term Scheduler
3. Medium Term Scheduler

Let's discuss all the different types of Schedulers in detail:

Long Term Scheduler

The long-term scheduler runs less frequently. Long Term Schedulers decide which program must get into the job queue. From the job queue, the Job Processor selects processes and loads them into the memory for execution. The primary aim of the Job Scheduler is to maintain a good degree of Multiprogramming. An optimal degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory.

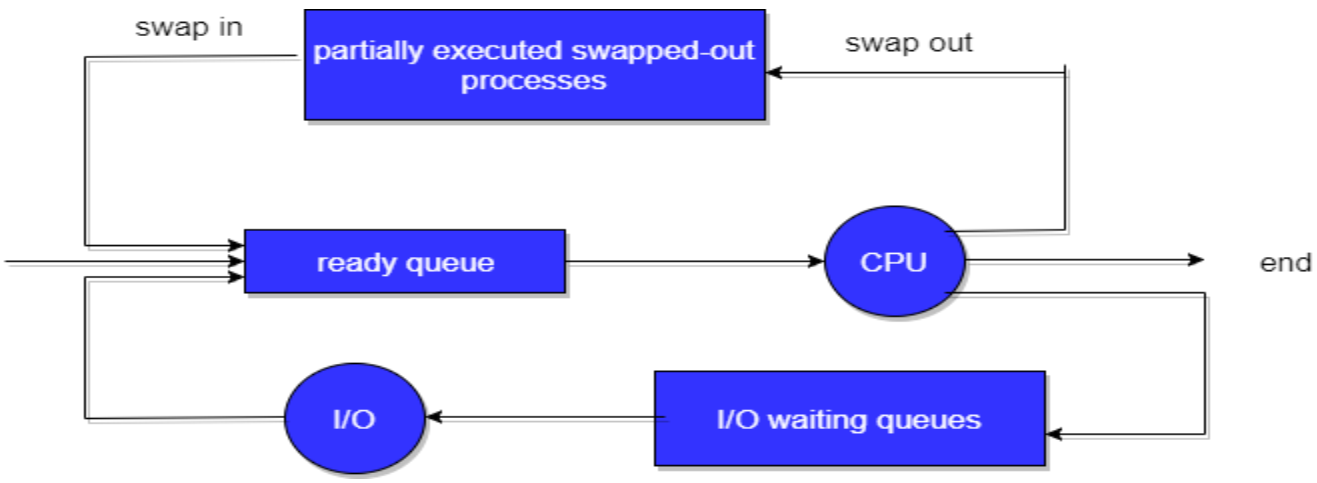
Short Term Scheduler

This is also known as CPU Scheduler and runs very frequently. The primary aim of this scheduler is to enhance CPU performance and increase process execution rate.

Medium Term Scheduler

This scheduler removes the processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out and is later swapped in, by the medium-term scheduler.

Swapping may be necessary to improve the process mix, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. This complete process is described in the below diagram:



Addition of Medium-term scheduling to the queueing diagram.

What is **Context Switch**?

1. Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a Context Switch.

2. The context of a process is represented in the Process Control Block(PCB) of a process; it includes the value of the CPU registers, the process state, and memory-management information. When a context switch occurs, the Kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

3. Context switch time is pure overhead because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds range from 1 to 1000 microseconds.

4. Context Switching has become such a performance bottleneck that programmers are using new structures(threads) to avoid it whenever and wherever possible.

Operations on Process

Two major operations are

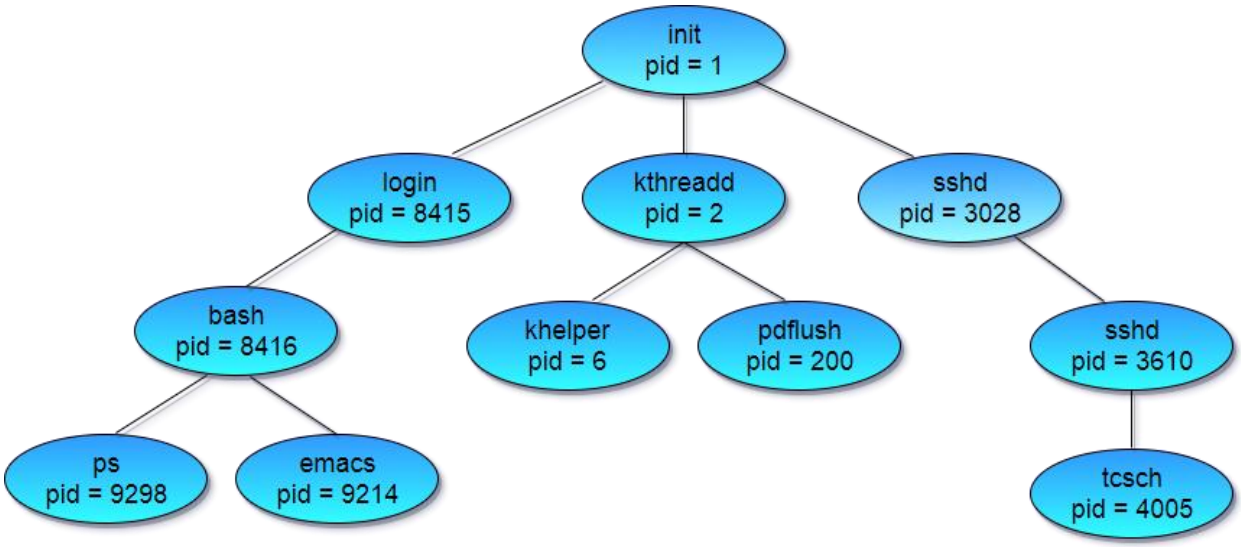
- **Process Creation**
- **Process Termination**.

Process Creation

Through appropriate system calls, such as fork or spawn, processes may create other processes. The process which creates other processes is termed the parent of the other process, while the created sub-process is termed its child.

Each process is given an integer identifier, termed a process identifier, or PID. The parent PID (PPID) is also stored for each process.

On a typical UNIX system, the process scheduler is termed as sched and is given PID 0. The first thing done by it at system start-up time is to launch init, which gives that process PID 1. Further Init launches all the system daemons and user logins, and becomes the ultimate parent of all other processes.



A child process may receive some amount of shared resources with its parent depending on system implementation. To prevent runaway children from consuming all of a certain system resource, child processes may or may not be limited to a subset of the resources originally allocated to the parent.

There are two options for the parent process after creating the child :

- Wait for the child process to terminate before proceeding. Parent process makes a wait() system call, for either a specific child process or for any particular child process, which causes the parent process to block until the wait() returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
- Run concurrently with the child, continuing to process without waiting. When a UNIX shell runs a process as a background task, this is the operation seen. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

To illustrate these different implementations, let us consider the UNIX operating system. In UNIX, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork system call, with one difference: The return code for the fork system call is zero for the new(child) process, whereas the(non zero) process identifier of the child is returned to the parent.

Typically, the execlp system call is used after the fork system call by one of the two processes to replace the process memory space with a new program. The execlp system call loads a binary file into memory - destroying the memory image of the program containing the execlp system call – and starts its execution. In this manner, the two processes can communicate, and then go their separate ways.

Below is a C program to illustrate forking a separate process using UNIX(made using Ubuntu):

```
#include<stdio.h>
void main(int argc, char *argv[])
{
   int pid;
   /* Fork another process */
   pid = fork();
   if(pid < 0)
   {
      //Error occurred
      fprintf(stderr, "Fork Failed");
      exit(-1);
   }
   else if (pid == 0)
   {
```

```
        //Child process
        execlp("/bin/ls","ls",NULL);
    }
    else
    {
        //Parent process
        //Parent will wait for the child to complete
        wait(NULL);
        printf("Child complete");
        exit(0);
    }
}
```

Process Termination

By exiting (system call), typically returning an int, processes may request their termination. This int is passed along to the parent if it is doing a wait(), and is typically zero on successful completion and some non-zero code in the event of any problem.

Processes may also be terminated by the system for a variety of reasons, including:

- The inability of the system to deliver the necessary system resources.
- In response to a KILL command or other unhandled process interrupts.
- A parent may kill their children if the task assigned to them is no longer needed i.e. if the need of having a child terminates.
- If the parent exits, the system may or may not allow the child to continue without a parent (In UNIX systems, orphaned processes are generally inherited by init, which then proceeds to kill them.)

When a process ends, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process already became an orphan.

The processes which are trying to terminate but cannot do so because their parent is not waiting for them are eventually inherited by init as orphans and killed off.

CPU Scheduling in Operating System

CPU scheduling is a process that allows one process to use the CPU while the execution of another process is on hold(in waiting for state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. CPU scheduling aims to make the system efficient, fast, and fair.

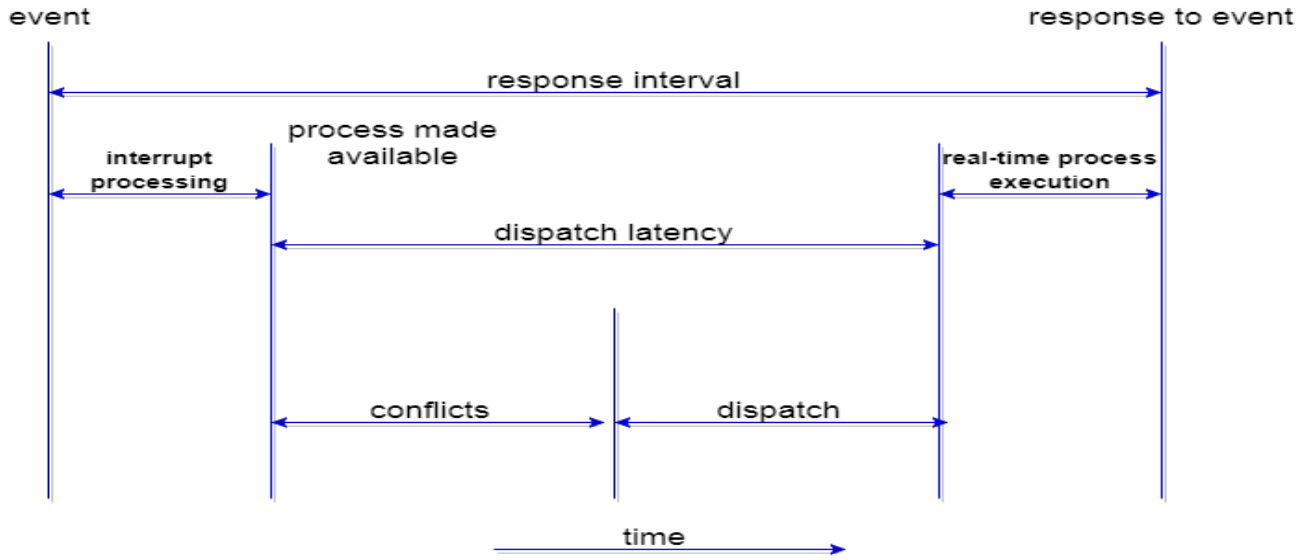
Whenever the CPU becomes idle, the operating system must select one of the processes in the **ready queue** to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute and allocates the CPU to one of them.

CPU Scheduling: Dispatcher

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:



Types of CPU 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 I/O request or 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, 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-pre-emptive; otherwise, the scheduling scheme is pre-emptive.

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 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 pre-emptive scheduling.

In non-preemptive scheduling, it does not interrupt a process running CPU in the middle of the execution. Instead, it waits till the process completes its CPU burst time, and then after that, it can allocate the CPU to any other process.

Some Algorithms based on non-preemptive scheduling are Shortest Job First (SJF non-pre-emptive) Scheduling and Priority (non- pre-emptive version) Scheduling, etc.

Process	Arrival time	CPU Burst Time (in millisecond
P0	2	8
P1	3	6
P2	0	9
P3	1	4

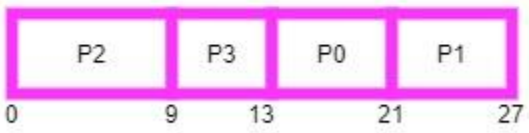


Figure: Non-Preemptive Scheduling

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.

Thus, this type of scheduling is used mainly when a process switches either from running state to ready state or from waiting for the state to ready state. The resources (that is CPU cycles) are mainly allocated to the process for a limited amount of time and they are taken away, and after that, the process is again placed back in the ready queue in the case that process still has a CPU burst time remaining. That process stays in the ready queue until it gets the next chance to execute.

Some Algorithms that are based on Preemptive scheduling are Round Robin Scheduling (RR), Shortest Remaining Time First (SRTF), Priority (Preemptive version) Scheduling, etc.

Process	Arrival time	CPU Burst Time (in millisecond
P0	2	3
P1	3	5
P2	0	6
P3	1	5

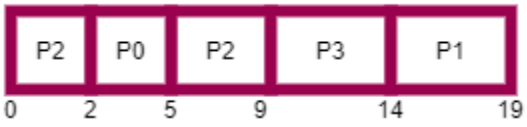


Figure: Preemptive Scheduling

CPU Scheduling: Scheduling Criteria

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

CPU Utilization

To make out the best use of the CPU and not to waste any CPU cycle, the CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

Throughput

It is the total number of processes completed per unit of time or rather says the total amount of work done in a unit of time. This may range from 10/seconds to 1/hour depending on the specific processes.

Turnaround Time

It is the amount of time taken to execute a particular process, i.e. The interval from the time of submission of the process to the time of completion of the process(Wall clock time).

Waiting Time

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

Load Average

It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

Response Time

The amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

Scheduling Algorithms

To decide which process to execute first and which process to execute last to achieve maximum CPU utilization, computer scientists have defined some algorithms, 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**
7. **Shortest Remaining Time First (SRTF)**
8. **Longest Remaining Time First (LRTF)**
9. **Highest Response Ratio Next (HRRN)**

First Come First Serve Scheduling

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.

Calculating Average Waiting Time

For every scheduling algorithm, Average waiting time is a crucial parameter to judge it's performance.

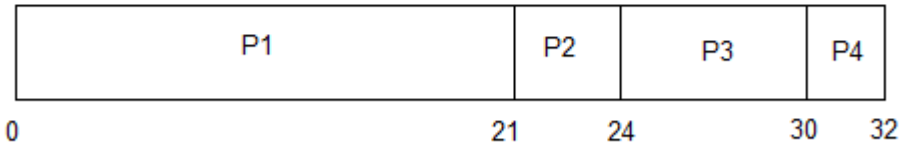
AWT or Average waiting time is the average of the waiting times of the processes in the queue, waiting for the scheduler to pick them for execution.

Lower the Average Waiting Time, better the scheduling algorithm.

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the FCFS scheduling algorithm.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



This is the GANTT chart for the above processes

The average waiting time will be 18.75 ms

For the above given proccesses, first **P1** will be provided with the CPU resources,

- Hence, waiting time for **P1** will be 0
- **P1** requires 21 ms for completion, hence waiting time for **P2** will be 21 ms
- Similarly, waiting time for process **P3** will be execution time of **P1** + execution time for **P2**, which will be (21 + 3) ms = 24 ms.
- For process **P4** it will be the sum of execution times of **P1**, **P2** and **P3**.

The **GANTT chart** above perfectly represents the waiting time for each process.

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.

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 */

#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;
```

output:

```
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.

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.
- 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)

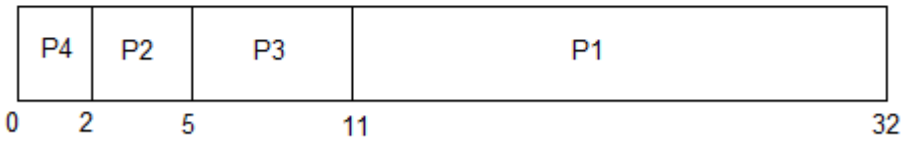
Non Pre-emptive Shortest Job First

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

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :



Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

As you can see in the **GANTT chart** above, the process **P4** will be picked up first as it has the shortest burst time, then **P2**, followed by **P3** and at last **P1**.

We scheduled the same set of processes using the First come first serve algorithm in the previous tutorial, and got average waiting time to be 18.75 ms, whereas with SJF, the average waiting time comes out 4.5 ms.

Problem with Non Pre-emptive SJF

If the arrival time for processes are different, which means all the processes are not available in the ready queue at time 0, and some jobs arrive after some time, in such situation, sometimes process with short burst time have to wait for the current process's execution to finish, because in Non Pre-emptive SJF, on arrival of a process with short duration, the existing job/process's execution is not halted/stopped to execute the short job first.

This leads to the problem of Starvation, where a shorter process has to wait for a long time until the current longer process gets executed. This happens if shorter jobs keep coming, but this can be solved using the concept of aging.

Pre-emptive Shortest Job First

In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with short burst time arrives, the existing process is preempted or removed from execution, and the shorter job is executed first.

The average waiting time will be,((5-3)+(6-2)+(12-1))/4=8.75

The average waiting time for preemptive shortest job first scheduling is less than both,non preemptive SJF scheduling and FCFS scheduling

As you can see in the GANTT chart above, as P1 arrives first, hence it's execution starts immediately, but just after 1 ms, process P2 arrives with a burst time of 3 ms which is less than the burst time of P1, hence the process P1(1 ms done, 20 ms left) is preemptied and process P2 is executed.

As P2 is getting executed, after 1 ms, P3 arrives, but it has a burst time greater than that of P2, hence execution of P2 continues. But after another millisecond, P4 arrives with a burst time of 2 ms, as a result P2(2 ms done, 1 ms left) is preemptied and P4 is executed.

After the completion of P4, process P2 is picked up and finishes, then P2 will get executed and at last P1.

The Pre-emptive SJF is also known as Shortest Remaining Time First, because at any given point of time, the job with the shortest remaining time is executed first.

Program for SJF Scheduling

In the below program, we consider the arrival time of all the jobs to be 0.

Also, in the program, we will sort all the jobs based on their burst time and then execute them one by one, just like we did in FCFS scheduling program.

```
// c++ program to implement Shortest Job first

#include<bits/stdc++.h>

using namespace std;

 struct Process

{

   int pid;     // process ID

   int bt;      // burst Time

};

 /*     this function is used for sorting all     processes in increasing order of burst time */

bool comparison(Process a, Process b)

{

    return (a.bt < b.bt);

}

// function to find the waiting time for all processes

void findWaitingTime(Process proc[], int n, int wt[])

{

    // waiting time for first process is 0
```

```
    wt[0] = 0;
     // calculating waiting time
    for (int i = 1; i < n ; i++)
    {
        wt[i] = proc[i-1].bt + wt[i-1] ;
    }
}
 // function to calculate turn around time
void findTurnAroundTime(Process proc[], int n, int wt[], int tat[])
{
    // calculating turnaround time by adding bt[i] + wt[i]
    for (int i = 0; i < n ; i++)
    {
        tat[i] = proc[i].bt + wt[i];
    }
}// function to calculate average time
void findAverageTime(Process proc[], int n)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
     // function to find waiting time of all processes
    findWaitingTime(proc, n, wt);
     // function to find turn around time for all processes
    findTurnAroundTime(proc, n, wt, tat);
     // display processes along with all details
    cout << "\nProcesses "<< " 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 << " " << proc[i].pid << "\t\t"
             << proc[i].bt << "\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 proc[] = {{1, 21}, {2, 3}, {3, 6}, {4, 2}};
    int n = sizeof proc / sizeof proc[0];
     // sorting processes by burst time.
    sort(proc, proc + n, comparison);
     cout << "Order in which process gets executed\n";
    for (int i = 0 ; i < n; i++)
    {
        cout << proc[i].pid <<" ";
    }
     findAverageTime(proc, n);
       return 0;
}
```

Output:

```
Order in which process gets executed
4 2 3 1
Processes  Burst time  Waiting time  Turn around time
 4          2          0             2
 2          3          2             5
 3          6          5             11
 1          21         11            32
Average waiting time = 4.5
Average turn around time = 12.5
```

Priority CPU Scheduling

In this tutorial we will understand the priority scheduling algorithm, how it works and its advantages and disadvantages.

In the Shortest Job First scheduling algorithm, the priority of a process is generally the inverse of the CPU burst time, i.e. the larger the burst time the lower is the priority of that process.

In case of priority scheduling the priority is not always set as the inverse of the CPU burst time, rather it can be internally or externally set, but yes the scheduling is done on the basis of priority of the process where the process which is most urgent is processed first, followed by the ones with lesser priority in order.

Processes with same priority are executed in FCFS manner.

The priority of process, when internally defined, can be decided based on memory requirements, time limits ,number of open files, ratio of I/O burst to CPU burst etc.

Whereas, external priorities are set based on criteria outside the operating system, like the importance of the process, funds paid for the computer resource use, market factor etc.

Types of Priority Scheduling Algorithm

Priority scheduling can be of two types:

1. **Preemptive Priority Scheduling**: If the new process arrived at the ready queue has a higher priority than the currently running process, the CPU is preempted, which means the processing of the current process is stoped and the incoming new process with higher priority gets the CPU for its execution.

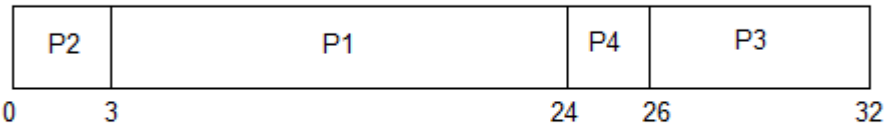
2. **Non-Preemptive Priority Scheduling**: In case of non-preemptive priority scheduling algorithm if a new process arrives with a higher priority than the current running process, the incoming process is put at the head of the ready queue, which means after the execution of the current process it will be processed.

Example of Priority Scheduling Algorithm

Consider the below table fo processes with their respective CPU burst times and the priorities.

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,



The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

As you can see in the GANTT chart that the processes are given CPU time just on the basis of the priorities.

Problem with Priority Scheduling Algorithm

In priority scheduling algorithm, the chances of indefinite blocking or starvation.

A process is considered blocked when it is ready to run but has to wait for the CPU as some other process is running currently.

But in case of priority scheduling if new higher priority processes keeps coming in the ready queue then the processes waiting in the ready queue with lower priority may have to wait for long durations before getting the CPU for execution.

In 1973, when the IBM 7904 machine was shut down at MIT, a low-priority process was found which was submitted in 1967 and had not yet been run.

Using Aging Technique with Priority Scheduling

To prevent starvation of any process, we can use the concept of aging where we keep on increasing the priority of low-priority process based on the its waiting time.

For example, if we decide the aging factor to be 0.5 for each day of waiting, then if a process with priority 20(which is comparatively low priority) comes in the ready queue. After one day of waiting, its priority is increased to 19.5 and so on.

Doing so, we can ensure that no process will have to wait for indefinite time for getting CPU time for processing.

Implementing Priority Scheduling Algorithm in C++

Implementing priority scheduling algorithm is easy. All we have to do is to sort the processes based on their priority and CPU burst time, and then apply FCFS Algorithm on it.

Here is the C++ code for priority scheduling algorithm:

```
// Implementation of Priority scheduling algorithm

#include<bits/stdc++.h>

using namespace std;

  struct Process

{          // this is the process ID

    int pid;

    // the CPU burst time

    int bt;

    // priority of the process

    int priority;

}; // sort the processes based on priority

bool sortProcesses(Process a, Process b)

{

    return (a.priority > b.priority);

}

 // Function to find the waiting time for all processes

void findWaitingTime(Process proc[], int n, int wt[])

{

    // waiting time for first process is 0

    wt[0] = 0;

      // calculating waiting time
```

```
    for (int  i = 1; i < n ; i++ )
        wt[i] =  proc[i-1].bt + wt[i-1] ;
}
  // Function to calculate turn around time
void findTurnAroundTime( Process proc[], int n,
                         int wt[], int tat[])
{
    // calculating turnaround time by adding
    // bt[i] + wt[i]
    for (int  i = 0; i < n ; i++)
        tat[i] = proc[i].bt + wt[i];
}  //Function to calculate average time
void findavgTime(Process proc[], int n)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
      //Function to find waiting time of all processes
    findWaitingTime(proc, n, wt);
      //Function to find turn around time for all processes
    findTurnAroundTime(proc, n, wt, tat);
      //Display processes along with all details
    cout << "\nProcesses  "<< " 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 << "   " << proc[i].pid << "\t\t"
```

```
            << proc[i].bt << "\t    " << wt[i]
            << "\t\t  " << tat[i] <<endl;
    }
     cout << "\nAverage waiting time = "
        << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
        << (float)total_tat / (float)n;
}
 void priorityScheduling(Process proc[], int n)
{     // Sort processes by priority
    sort(proc, proc + n, sortProcesses);
     cout<< "Order in which processes gets executed \n";
    for (int  i = 0 ; i <  n; i++)
       cout << proc[i].pid <<" " ;
     findavgTime(proc, n);
} // Driver code
int main()
{
    Process proc[] = {{1, 10, 2}, {2, 5, 0}, {3, 8, 1}};
    int n = sizeof proc / sizeof proc[0];
    priorityScheduling(proc, n);
    return 0;
}
```

Round Robin Scheduling

Round Robin(RR) scheduling algorithm is mainly designed for time-sharing systems. This algorithm is similar to FCFS scheduling, but in Round Robin(RR) scheduling, preemption is added which enables the system to switch between processes.

- A fixed time is allotted to each process, called a **quantum**, for execution.

- Once a process is executed for the given time period that process is preempted and another process executes for the given time period.
- Context switching is used to save states of preempted processes.
- This algorithm is simple and easy to implement and the most important is thing is this algorithm is starvation-free as all processes get a fair share of CPU.
- It is important to note here that the length of time quantum is generally from 10 to 100 milliseconds in length.

Some important characteristics of the Round Robin(RR) Algorithm are as follows:

1. Round Robin Scheduling algorithm resides under the category of Preemptive Algorithms.
2. This algorithm is one of the oldest, easiest, and fairest algorithm.
3. This Algorithm is a real-time algorithm because it responds to the event within a specific time limit.
4. In this algorithm, the time slice should be the minimum that is assigned to a specific task that needs to be processed. Though it may vary for different operating systems.
5. This is a hybrid model and is clock-driven in nature.
6. This is a widely used scheduling method in the traditional operating system.

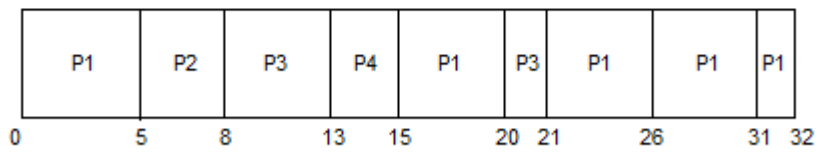
Important terms

1. **Completion Time** It is the time at which any process completes its execution.
2. **Turn Around Time** This mainly indicates the time Difference between completion time and arrival time. The Formula to calculate the same is: **Turn Around Time = Completion Time – Arrival Time**
3. **Waiting Time(W.T):** It Indicates the time Difference between turn around time and burst time. And is calculated as **Waiting Time = Turn Around Time – Burst Time**

Let us now cover an example for the same:

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

In the above diagram, arrival time is not mentioned so it is taken as 0 for all processes.

Note: If arrival time is not given for any problem statement then it is taken as 0 for all processes; if it is given then the problem can be solved accordingly.

Explanation

The value of time quantum in the above example is 5.Let us now calculate the Turn around time and waiting time for the above example :

Processes	**Burst Time**	**Turn Around Time** **Turn Around Time = Completion Time – Arrival Time**	**Waiting Time** **Waiting Time = Turn Around Time – Burst Time**
P1	21	32-0=32	32-21=11
P2	3	8-0=8	8-3=5
P3	6	21-0=21	21-6=15
P4	2	15-0=15	15-2=13

Average waiting time is calculated by adding the waiting time of all processes and then dividing them by no.of processes.

average waiting time = waiting time of all processes/ no.of processes

average waiting time=11+5+15+13/4 = 44/4= **11ms**

```
// Program implementation in C++ for Round Robin scheduling

#include<iostream>

using namespace std;

//The Function to find the waiting time for all processes

void fWaitingTime(int processes[], int n,

                        int bt[], int wt[], int quantum)

{

    // Let us Make a copy of burst times bt[] to store remaining burst times

        int rem_bt[n];

    for (int i = 0 ; i < n ; i++)

        rem_bt[i] = bt[i];

    int t = 0; // for Current time

    // Let us keep traverse the processes in the round robin manner until all of them are not done.

    while (1)
```

```
        {
                bool done = true;
                //let us Traverse all processes one by one repeatedly
                for (int i = 0 ; i < n; i++)
                {       // If burst time of a process is greater than 0 then there is a need to process further
                        if (rem_bt[i] > 0)
                        {
                                done = false; // indicates there is a pending process
                                if (rem_bt[i] > quantum)
                                {
// By Increasing the value of t it shows how much time a process has been processed
                                        t += quantum;
// Decreasing the burst_time of current process by the quantum
                                        rem_bt[i] -= quantum;
                                }
// If burst time is smaller than or equal to the quantum then it is Last cycle for this process
                                else
                                {
// Increase the value of t to show how much time a process has been processed
                                        t = t + rem_bt[i];
// Waiting time is current time minus time used by this process.
                                        wt[i] = t - bt[i];
// As the process gets fully executed thus remaining burst time becomes 0.
                                        rem_bt[i] = 0;
                                }
                        }
                }
// If all the processes are done
                if (done == true)
```

```
                break;
        }
}
// Function used to calculate the turn around time
void fTurnAroundTime(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 the average time
void findavgTime(int processes[], int n, int bt[],int quantum)
{
        int wt[n], tat[n], total_wt = 0, total_tat = 0;
        // Function to find waiting time of all processes
        fWaitingTime(processes, n, bt, wt, quantum);
        // Function to find turn around time for all processes
        fTurnAroundTime(processes, n, bt, wt, tat);
        // Display processes along with all details
        cout << "Processes "<< " Burst time "
                << " Waiting time " << " Turn around time\n";
        // Calculate the 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;
}
//Given below is the Driver Code
int main()
{       // process id's
        int processes[] = { 1, 2, 3,4};
        int x = sizeof processes / sizeof processes[0];
        // Burst time of all processes
        int burst_time[] = {21, 13, 6,12};
        // Time quantum
        int quantum = 2;
        findavgTime(processes, x, burst_time, quantum);
        return 0;
}
```

Output

The output of the above code is as follows:

```
Processes  Burst time  Waiting time  Turn around time
 1              21          31              52
 2              13          32              45
 3              6           16              22
 4              12          30              42
Average waiting time = 27.25
Average turn around time = 40.25
```

Advantages of Round Robin Scheduling Algorithm

Some advantages of the Round Robin scheduling algorithm are as follows:

- While performing this scheduling algorithm, a particular time quantum is allocated to different jobs.
- In terms of average response time, this algorithm gives the best performance.

- With the help of this algorithm, all the jobs get a fair allocation of CPU.
- In this algorithm, there are no issues of starvation or convoy effect.
- This algorithm deals with all processes without any priority.
- This algorithm is cyclic in nature.
- In this, the newly created process is added to the end of the ready queue.
- Also, in this, a round-robin scheduler generally employs time-sharing which means providing each job a time slot or quantum.
- In this scheduling algorithm, each process gets a chance to reschedule after a particular quantum time.

Disadvantages of Round Robin Scheduling Algorithm

Some disadvantages of the Round Robin scheduling algorithm are as follows:

- This algorithm spends more time on context switches.
- For small quantum, it is time-consuming scheduling.
- This algorithm offers a larger waiting time and response time.
- In this, there is low throughput.
- If time quantum is less for scheduling then its Gantt chart seems to be too big.

Some Points to Remember

1.Decreasing value of Time quantum

With the decreasing value of time quantum

- The number of context switches increases.
- The Response Time decreases
- Chances of starvation decrease in this case.

For the **smaller value of time quantum,** it becomes better in terms of **response time.**

2.Increasing value of Time quantum

With the increasing value of time quantum

- The number of context switch decreases
- The Response Time increases
- Chances of starvation increases in this case.

For the higher value of time quantum, it becomes better in terms of the **number of the context switches.**

3. If the value of **time quantum is increasing** then Round Robin Scheduling tends to **become FCFS Scheduling.**

4.In this case, when the value of time quantum **tends to infinity** then the Round Robin Scheduling **becomes FCFS Scheduling.**

5. Thus the performance of Round Robin scheduling mainly depends on the **value of the time quantum.**

6.And the value of the **time quantum** should be such that it is neither **too big nor too small.**

Multilevel Queue Scheduling Algorithm

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.

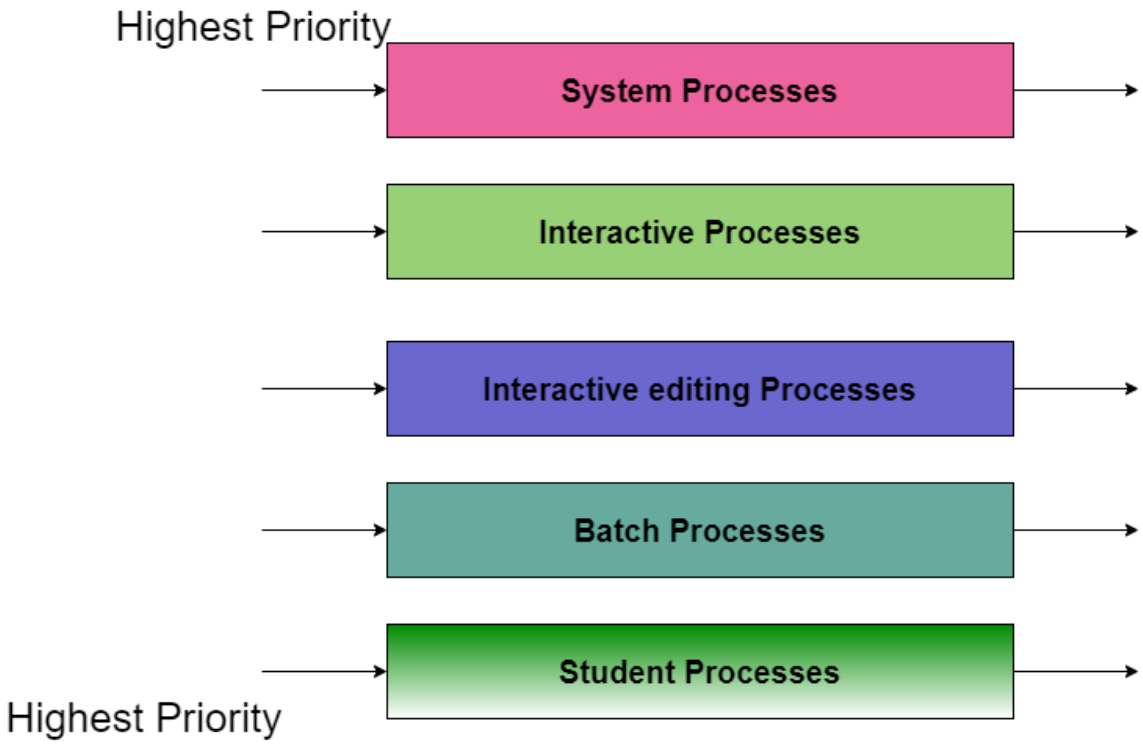
For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by the Round Robin algorithm, while the background queue is scheduled by an FCFS algorithm.

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.



In this case, if there are no processes on the higher priority queue only then the processes on the low priority queues will run.

For Example: Once processes on the system queue, the Interactive queue, and Interactive editing queue become empty, only then the processes on the batch queue will run.

The Description of the processes in the above diagram is as follows:

- **System Process** The Operating system itself has its own process to run and is termed as System Process.
- **Interactive Process** The Interactive Process is a process in which there should be the same kind of interaction (basically an online game).
- **Batch Processes** Batch processing is basically a technique in the Operating system that collects the programs and data together in the form of the batch before the processing starts.
- **Student Process** The system process always gets the highest priority while the student processes always get the lowest priority.

In an operating system, there are many processes, in order to obtain the result we cannot put all processes in a queue; thus this process is solved by Multilevel queue scheduling.

Implementation

Given below is the C implementation of Multilevel Queue Scheduling:

```
#include<stdio.h>

int main()

{

    int p[20],bt[20], su[20], wt[20],tat[20],i, k, n, temp;

    float wtavg, tatavg;

    printf("Enter the number of processes:");

    scanf("%d",&n);

    for(i=0;i<n;i++)

    {

        p[i] = i;

        printf("Enter the Burst Time of Process%d:", i);

        scanf("%d",&bt[i]);

        printf("System/User Process (0/1) ? ");

        scanf("%d", &su[i]);

    }

    for(i=0;i<n;i++)
```

```
		for(k=i+1;k<n;k++)
			if(su[i] > su[k])
			{
			temp=p[i];
			p[i]=p[k];
			p[k]=temp;
			temp=bt[i];
			bt[i]=bt[k];
			bt[k]=temp;
			temp=su[i];
			su[i]=su[k];
			su[k]=temp;
			}
	wtavg = wt[0] = 0;
	tatavg = tat[0] = bt[0];
	for(i=1;i<n;i++)
	{
		wt[i] = wt[i-1] + bt[i-1];
		tat[i] = tat[i-1] + bt[i];
		wtavg = wtavg + wt[i];
		tatavg = tatavg + tat[i];
	}
printf("\nPROCESS\t\t SYSTEM/USER PROCESS \tBURST TIME\tWAITING TIME\tTURNAROUND TIME");
	for(i=0;i<n;i++)
		printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],su[i],bt[i],wt[i],tat[i]);
	printf("\nAverage Waiting Time is --- %f",wtavg/n);
	printf("\nAverage Turnaround Time is --- %f",tatavg/n);
	return 0;
```

}

Output

The output of the above code is as follows:

```
Enter the number of processes:3
Enter the Burst Time of Process0:12
System/User Process (0/1) ? 0
Enter the Burst Time of Process1:18
System/User Process (0/1) ? 0
Enter the Burst Time of Process2:15
System/User Process (0/1) ? 1

PROCESS          SYSTEM/USER PROCESS    BURST TIME      WAITING TIME    TURNAROUND TIME
0                0                 12              0                12
1                0                 18              12               30
2                1                 15              30               45
Average Waiting Time is --- 14.000000
Average Turnaround Time is --- 29.000000
```

Advantages of Multilevel Queue Scheduling

With the help of this scheduling we can apply various kind of scheduling for different kind of processes:

For System Processes: First Come First Serve(FCFS) Scheduling.

For Interactive Processes: Shortest Job First (SJF) Scheduling.

For Batch Processes: Round Robin(RR) Scheduling

For Student Processes: Priority Scheduling

Disadvantages of Multilevel Queue Scheduling

The main disadvantage of Multilevel Queue Scheduling is the problem of starvation for lower-level processes.

Starvation:

Due to starvation lower-level processes either never execute or have to wait for a long amount of time because of lower priority or higher priority process taking a large amount of time.

Multilevel Feedback Queue Scheduling

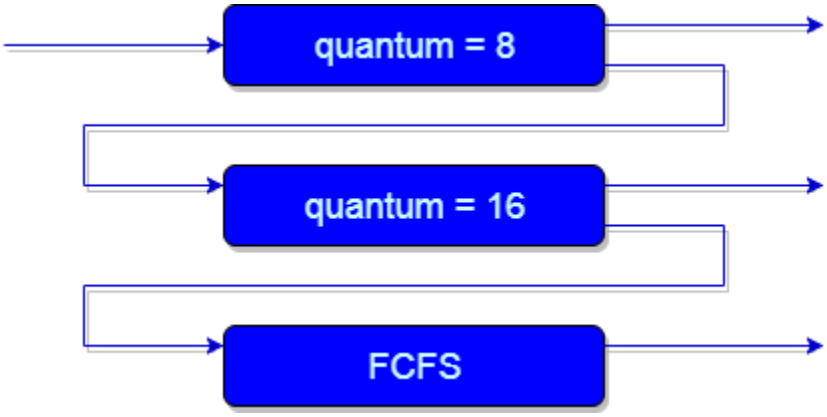
In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.

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.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues.
- The scheduling algorithm for each queue.
- The method used to determine when to upgrade a process to a higher-priority queue.
- The method used to determine when to demote a process to a lower-priority queue.
- The method used to determine which queue a process will enter when that process needs service.

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**.**



An example of a multilevel feedback queue can be seen in the above figure.

Explanation:

First of all, Suppose that queues 1 and 2 follow round robin with time quantum 8 and 16 respectively and queue 3 follows FCFS. One of the implementations of Multilevel Feedback Queue Scheduling is as follows:

1. If any process starts executing then firstly it enters queue 1.
2. In queue 1, the process executes for 8 unit and if it completes in these 8 units or it gives CPU for I/O operation in these 8 units unit than the priority of this process does not change, and if for some reasons it again comes in the ready queue than it again starts its execution in the Queue 1.
3. If a process that is in queue 1 does not complete in 8 units then its priority gets reduced and it gets shifted to queue 2.
4. Above points 2 and 3 are also true for processes in queue 2 but the time quantum is 16 units. Generally, if any process does not complete in a given time quantum then it gets shifted to the lower priority queue.
5. After that in the last queue, all processes are scheduled in an FCFS manner.
6. It is important to note that a process that is in a lower priority queue can only execute only when the higher priority queues are empty.
7. Any running process in the lower priority queue can be interrupted by a process arriving in the higher priority queue.

Also, the above implementation may differ for the example in which the last queue will follow **Round-robin Scheduling.**

In the above Implementation, there is a problem and that is; Any process that is in the lower priority queue has to suffer starvation due to some short processes that are taking all the CPU time.

And the solution to this problem is: There is a solution that is to boost the priority of all the process after regular intervals then place all the processes in the highest priority queue.

The need for Multilevel Feedback Queue Scheduling (MFQS)

Following are some points to understand the need for such complex scheduling:

- This scheduling is more flexible than Multilevel queue scheduling.
- This algorithm helps in reducing the response time.
- In order to optimize the turnaround time, the SJF algorithm is needed which basically requires the running time of processes in order to schedule them. As we know that the running time of processes is not known in advance. Also, this scheduling mainly runs a process for a time quantum and after that, it can change the priority of the process if the process is long. Thus this scheduling algorithm mainly learns from the past behavior of the processes and then it can predict the future behavior of the processes. In this way, MFQS tries to run a shorter process first which in return leads to optimize the turnaround time.

Advantages of MFQS

- This is a flexible Scheduling Algorithm
- This scheduling algorithm allows different processes to move between different queues.
- In this algorithm, A process that waits too long in a lower priority queue may be moved to a higher priority queue which helps in preventing starvation.

Disadvantages of MFQS

- This algorithm is too complex.
- As processes are moving around different queues which leads to the production of more CPU overheads.
- In order to select the best scheduler this algorithm requires some other means to select the values

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.

Shortest Job First (SJF)

Advantages:

- 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.

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)

Advantages: 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.

Priority based Scheduling

Advantages :

- 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.

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.

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.

For example, the scheduler itself is a kernel based process, it should run first so that it can schedule other processes.