UNIT 2

CPU SCHEDULING

Today Operating System can run multiple processes. However, we know that only one process can be executed at the same time. Then to arrange and control the numerous processes in the system and utilize the CPU efficiently, some scheduling algorithm used by CPU. CPU Scheduling Algorithm primarily used in multiprogramming operating system. To execute a process in the simple system, a process required Input-Output devices, resources and CPU time. If the process is dealing with I/O devices, the CPU will sit idle. To overcome this problem and save time, OS manage the system in this way that if one process is busy with I/O devices, then another process will use CPU for execution of their process. So that management of time will achieve in a right way.

Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There is many processes in the ready queue to execute their process, scheduler select one of them for execution by CPU. A process should be a proper contrast to I/O bound and CPU bound. I/O required processes require more time for I/O than computation.

FACTORS AFFECTING THE SCHEDULING

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

Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, then another CPU burst, then another I/O burst, and so on. Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst

- **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 from among the processes in memory that are ready to execute, and allocates the CPU to one of them. The ready queue is not necessarily a first-in, first-out (FIFO) queue. A ready queue may 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.

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.

PREEMPTIVE AND NON-PREEMPTIVE SCHEDULING

Non-Pre-emptive Scheduling

Under non-pre-emptive 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 pre-emptive scheduling.

Pre-emptive 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.

CPU SCHEDULING CRITERIA

Different CPU-scheduling algorithms have different properties and may favour one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. There are many different criteria's to check when considering the "best" scheduling algorithm, they are:

- **CPU Utilization**

To make out the best use of CPU and not to waste any CPU cycle, 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 time or rather say total amount of work done in a unit of time. This may range from 10/second 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 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**

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

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

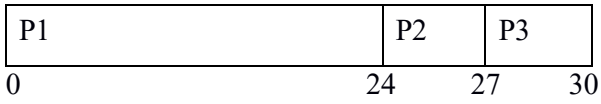
- **FIRST-COME, FIRST-SERVED SCHEDULING (FCFS)**

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 average waiting time under the FCFS policy, however, is often quite long.

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

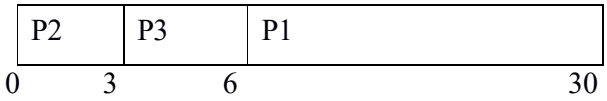
Process	Burst Time
P1	24
P2	3
P3	3

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following Gantt chart:



The waiting time is 0 milliseconds for process P_1, 24milliseconds for process P_2, and 27 milliseconds for process P_3. Thus, the average waiting time is (0 + 24 +27)/3 = 17milliseconds. The turn around time is 24 milliseconds for process P_1, 27milliseconds for process P_2, and 30 milliseconds for process P_3. Thus, the average turn around time is (24 + 27 +30)/3 = 27milliseconds.

If the processes arrive in the order P_2, P_3, P_1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds.

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

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

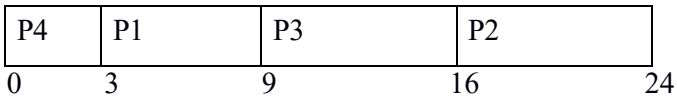
- **SHORTEST-JOB-FIRST SCHEDULING**

A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie. Note that a more appropriate term would be the shortest next CPU burst, because the scheduling is done by examining the length of the next CPU burst of a process, rather than its total length.

As an example, consider the following set of processes, with the length of the burst time given in milliseconds:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

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



The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2, 9 milliseconds for process P_3, and 0 milliseconds for process P_4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. If we were using the FCFS scheduling scheme, then the average waiting time could be 10.25 milliseconds.

The SJF scheduling algorithm is probably optimal, in that it gives the minimum average waiting time for a given set of processes. By moving a short process before a long one, the waiting time of the short process decreases 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. For long-term (or job) scheduling in a batch system, we can use as the length the process time limit that a user specifies when he submits the job. Thus, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short term CPU scheduling. There is no way to know the length of the next CPU burst. One

approach is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value.

The SJF algorithm may be either preemptive or non-pre-emptive. The choice arises when a new process arrives at the ready queue while a previous process is executing. The new process may have a shorter next CPU burst than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.

As an example, consider the following four processes, with the length of the CPU-burst time given in milliseconds.

Process	**Arrival Time**	Burst Time
P1	**0**	8
P2	**1**	4
P3	**2**	9
P4	**3**	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted the following Gantt chart:

P1	P2	P4	P1	P3
0	1	5	10	17 26

Process P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process is preempted, and process P_2 is scheduled. The average waiting time for this example is ((10 - 0) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4= 6.5 milliseconds.

A non pre-emptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

Example: Non-Preemptive SJF (varied arrival times)

Process	Arrival Time	Burst Time
P1	2	6
P2	1	8
P3	0	7
P4	3	3

P3	P4	P1	P2
0	7	10	16	24

Average waiting time = ((10–2)+(16–1)+(0–0)+(7–3))/4 = (8 + 15 + 0 + 4)/4 = 6.75

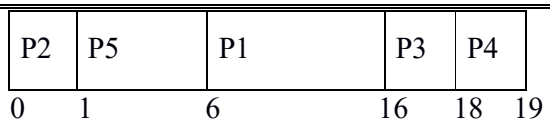
PRIORITY SCHEDULING

The SJF algorithm is a special case of the general priority-scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. 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. Priorities are generally 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. This difference can lead to confusion. In this we use low numbers to represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order P_1, P_2…..P_5 with the length of the CPU-burst time given in milliseconds:

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

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



Average waiting time is 8.2 milliseconds (i.e. (6+0+16+18+1)/5).

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

A major problem with priority-scheduling algorithms is indefinite blocking (or starvation). A process that is ready to run but lacking the CPU can be considered blockedwaiting for the CPU. A priority-scheduling algorithm can leave some low-priority processes waiting indefinitely for the CPU. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. A solution to the problem of indefinite blockage of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

- **ROUND-ROBIN SCHEDULING**

The Round-Robin (RR) scheduling algorithm is designed especially for time sharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum (or time slice),is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will

be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy, however, is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time
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 P_2 Since process P_2 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 P_1 for an additional time quantum. The resulting RR schedule is

P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 10 14 18 22 26 30

The average waiting time is 17/3 = 5.66milliseconds.

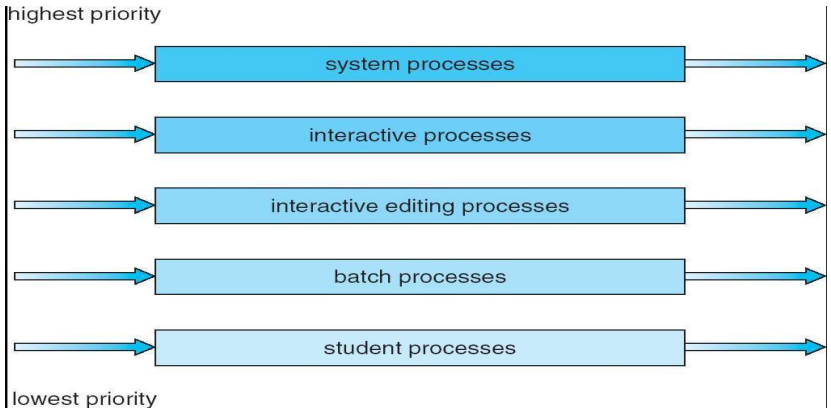
In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a process CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is preemptive. The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is very large (infinite), the RR policy is the same as the FCFS policy.

MULTILEVEL QUEUE SCHEDULING

A multilevel 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 an RR 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.



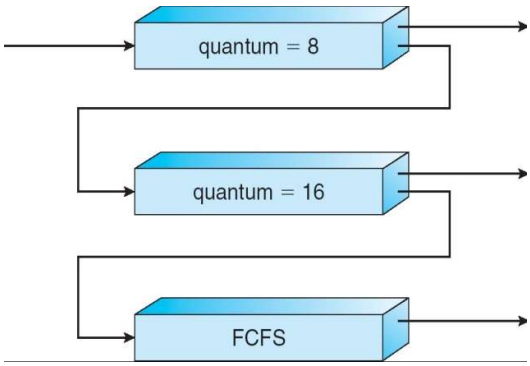
Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be pre-empted.

MULTILEVEL FEEDBACK QUEUE SCHEDULING

Normally, in a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system. Processes do not move between queues. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature.

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. This scheme leaves I/O bound and interactive processes in the higher-priority queues. 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.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2. The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process that arrives for queue a will, in turn, preempt a process in queue 1. process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16milliseconds. If it does not complete, it is pre-empted and is put into queue 2. Processes in queue 2 are run on an FCFS basis, only when queues 0 and 1 are empty.



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

- 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