Scheduling refers to the set of policies and mechanisms that an OS supports for determining the order of execution of the pending jobs and processes. A Scheduler is an operating system program (module) that selects the next job to be admitted for execution.

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process based on a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

**Scheduling Criteria**

A Scheduler algorithm is evaluated against some widely accepted performance criteria.

1. **C.P.U. Utilization:** It is defined as average fraction of time during which CPU is busy, executing either user programs or system modules. The key idea is that if the CPU is busy all the time then the utilization factor of all the components of the system will also be high.
2. **Throughput:**  It is defined as the average amount of work completed per unit time. One way to measure throughput is by means of the number of processes that are completed in a unit of time. Please note here that the higher is the number of processes, the more work is apparently being done by the system. Also note that the higher is the throughput better it is. But this approach is not very useful for comparison because this is dependent on the characteristics and the resource requirements of the process being executed. Thus, to compare the throughput of several scheduling algorithms it should feed the process with similar requirements.
3. **Turn around Time (TAT):** It is defined as the total time elapsed from the time the job is submitted (or process is created) to the time the job (or process) is completed. It is the sum of periods spend waiting to get into memory. Waiting in the ready queue, CPU time and I/O operations. So, we can write that –

**Turn around Time (TAT) = (Process Finish Time – Process Arrival Time)**

Also, note that the lower is the average turnaround time, better it is.

1. **Waiting Time (WT):** It is defined as the total time spent by the job (or process) while waiting in suspended state or ready state, in a multiprogramming environment. So, it is given as

**Waiting Time (WT) = (Turn Around Time – Processing Time)**

Please note that the lower is the average waiting time, better it is.

1. **Response Time (RT):** This parameter is usually considered for two systems – time sharing and real time operating system. However, its characteristics differ in these two systems. In time sharing system it may be defined as the interval from the time and last character of a command line of a program or transaction is entered to the time the last result appears on the terminal.

**Preemptive scheduling** is used when a process switches from running state to ready state or from the waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in the ready queue till it gets its next chance to execute.

Algorithms based on preemptive scheduling are: Round Robin (RR), Shortest Remaining Time First (SRTF), Priority (preemptive version), etc.

**Non-preemptive Scheduling** is used when a process terminates, or a process switch from running to the waiting state. In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state. In the case of non-preemptive scheduling 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 it can allocate the CPU to another process.

Algorithms based on non-preemptive scheduling are: Shortest Job First (SJF basically non preemptive) and Priority (non-preemptive version), etc.

|  |  |
| --- | --- |
| **Preemptive scheduling** | **Non-preemptive Scheduling** |
| In non-preemptive scheduling, if once a process has been allocated CPU, then the CPU cannot be taken away from that process | In Pre-emptive scheduling, the CPU can be taken away before the completion of the process |
| No Preference is given when a higher priority job comes | It is useful when a higher priority job comes as here the CPU can be snatched from a lower priority process. |
| The treatment of all processes is fairer. | The treatment of all processes is not fairer as CPU snatching is done either due to time constraints or due to a higher priority process requests for its execution |
| It is cheaper scheduling method.  For example: FCFS CPU Scheduling algorithm is non-preemptive. | It is costlier scheduling method.  For example: Round-Robin (RR) CPU scheduling algorithm is preemptive. |

**Key Differences Between Preemptive and Non-Preemptive Scheduling:**

1. In preemptive scheduling, the CPU is allocated to the processes for a limited time whereas, in non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to the waiting state.
2. The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and waits till its execution.
3. In Preemptive Scheduling, there is the overhead of switching the process from the ready state to running state, vise-verse and maintaining the ready queue. Whereas in the case of non-preemptive scheduling has no overhead of switching the process from running state to ready state.
4. In preemptive scheduling, if a high-priority process frequently arrives in the ready queue then the process with low priority must wait for a long, and it may have to starve. in the non-preemptive scheduling, if CPU is allocated to the process having a larger burst time then the processes with small burst time may have to starve.
5. Preemptive scheduling attains flexibility by allowing the critical processes to access the CPU as they arrive at, the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.
6. Preemptive Scheduling must maintain the integrity of shared data that’s why it is cost associative which is not the case with Non-preemptive Scheduling.

**First-Come, First-Served Scheduling (FCFS)**

It is the simplest of all the scheduling algorithms. **The basic Principle of this algorithm is to allocate the CPU in the order in which the process arrive.** Its implementation involves a ready queue that works in First-in First-out order. When the CPU is free, it is assigned to a process which is in front of the ready queue. Once this process goes into a running state, its PCB is removed from the queue. A FCFS CPU Scheduling algorithms is **non-preemptive** because the CPU has been allocated to a process that keeps the CPU busy until it is released. This usually results in poor performance. It is because of this non-preemption only that the component utilization rate is low and the system throughout is less. Also note here that the shorter jobs may suffer considerable turnaround delays and waiting times when CPU has been allocated to longer jobs.

**Advantages of FCFS algorithms**

1. It is simple algorithm and easy to implement.
2. It is suitable for Batch systems.

**Disadvantages of FCFS algorithms**

1. The average waiting time is not minimal. Under this scheme, the process that requests the CPU first is allocated the CPU first. When the process enters the ready queue, its Process Control Block (PCB) is linked which keeps the track of all the processes running and waiting. So, the average waiting time under the FCFS policy is very long.
2. It is not suitable for time sharing systems like Unix because in Unix, each user should get the CPU for an equal amount of time interval and this algorithm does not follow this principle.
3. A proper mix of CPU bound and I/O bound jobs is required to achieved good results from FCFS scheduling.

**Shortest Job First Scheduling (SJF)**

**The basic principle of this algorithm is to allocate the CPU to the process with least CPU-burst time.**  The processes are available in the ready queue. CPU is always assigned to the process with least CPU burst time requirement. **Please note that if the two processes have same CPU burst time, then FCFS is used to break the tie.** This algorithm can be either preemptive or non-preemptive.

**In non-preemptive scheduling, once the CPU cycle is allocated to process, the process holds it till it reaches a waiting state or terminated.**

**In Preemptive SJF Scheduling, jobs are put into the ready queue as they come. A process with shortest burst time begins execution. If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle.**

**Advantages of SJF Scheduling**

1. Used for long-term scheduling.
2. Reduces average waiting time.
3. Helpful for batch-type processing where runtimes are known in advance.
4. For long-term scheduling, we can obtain a burst time estimate from the job description.
5. It is necessary to predict the value of the next burst time for Short-Term Scheduling.
6. Optimal with regard to average turnaround time.
7. Since this algorithm gives minimum average waiting time so it is an optimal algorithm

**Disadvantages of SJF Scheduling**

1. It is necessary to know the job completion time beforehand as it is hard to predict.
2. Used for long-term scheduling in a batch system.
3. Can’t implement this algorithm for CPU scheduling for the short term as we can’t predict the length of the upcoming CPU burst. It is difficult to know the length of next CPU burst time.
4. Cause extremely long turnaround times or starvation.
5. Knowledge about the runtime length of a process is necessary.
6. Recording the elapsed time results in increased overhead on the processor.
7. It cannot be implemented at the level of short-term CPU Scheduling. Big jobs are waiting for CPU. This results in aging problem.

**Shortest Remaining Time First (SRTF)**

This algorithm is a preemptive scheduling algorithm. The short-term scheduler always chooses the process that has the shortest remaining processing time. When a new process joins the ready queue, the short-term scheduler compares the remaining time of executing process and new process. If the new process has the least CPU burst time, the scheduler selects that job and allocates CPU else it continues with the old process.

**Round Robin (RR) Scheduling algorithm**

This is one of the oldest, simplest and widely used algorithms. The basic principle of this algorithm is as follows

**Principle – “If there are n-processes in a ready queue and the time quantum is ‘q’ time intervals, then each process get ‘1/n’ of the CPU time in the chunks of at most ‘q’ units of time. Each process will have to wait for (n-1) x q time units until its next time quantum comes in”**

RR algorithm is like FCFS algorithm but not it is preemptive FCFS scheduling. The round robin scheduling algorithm is primarily used in time-sharing and a multi-user system where the primary requirement is to provide reasonably good response times and in general to share the system fairly among all system users. The preemption takes place after a fixed interval of time called the **time quantum or time slice**. The CPU time is divided into slices. Each process is allocated to small time (usually 10-100 milliseconds) while it executes.

**Please note that no process can run for more than one time slice because some other processes are awaiting in the ready queue.** If the time quantum expires then process goes to the end of the ready queue to wait for the next allocation. **Also note that if the running process releases a control to OS due to some I/O, then another process is scheduled to run.**

**RR** Scheduling algorithm utilizes the system resources uniformly. Small process may be executed in a single time-slice giving good response time whereas long processes may require several time slices and thus be forced to pass through ready queue a few times before completion.

The Performance of this **Pure Preemptive algorithm** depends on many factors –

1. **Size of time quantum (q):**

**If q is large,** then this algorithm becomes same as FCFS algorithm and thus performance degrades. **If q is small,** then the number of context switches increases and q almost equals the time taken to switch the CPU from one process to another. This wastes 50% of overall time. So, q should not be very large and should not be very small also so as to achieve good system performance.

1. **Number of context switches:**

The number of context switches should be too many as it slows down the overall execution of all the processes. **Please remember that the time quantum (q) should be large with respect to the context switch time.** This is done to ensure that a process keeps CPU for a considerable amount of time as compared to the time spent in context switching.

**Advantages:**

* Every process gets an equal share of the CPU.
* RR is cyclic in nature, so there is no starvation.

**Disadvantages:**

* Setting the quantum too short increases the overhead and lowers the CPU efficiency but setting it too long may cause a poor response to short processes.
* The average waiting time under the RR policy is often long.
* If time quantum is very high, then RR degrades to FCFS.

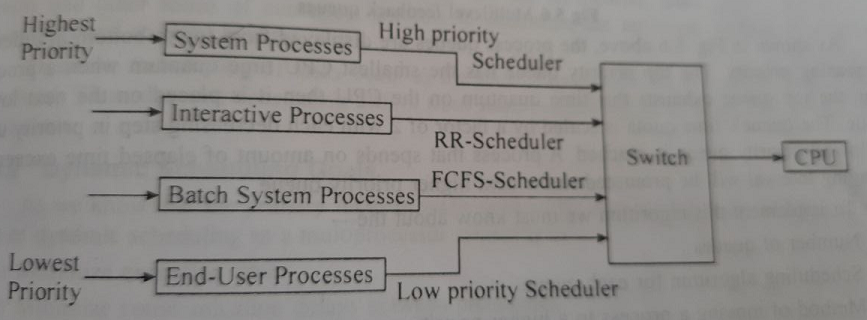
**Priority Scheduling (Preemptive and Non-Preemptive)** Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority. The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority depends upon memory requirements, time requirements, etc.

A CPU algorithm that schedules processes based on priority. It used in Operating systems for performing batch processes. If two jobs having the same priority are READY, it works on a FIRST COME, FIRST SERVED basis. In priority scheduling, a number is assigned to each process that indicates its priority level. Lower the number, higher is the priority. In this type of scheduling algorithm, if a newer process arrives, that is having a higher priority than the currently running process, then the currently running process is preempted.

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy, will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That’s because it doesn’t need special hardware (for example, a timer) like preemptive scheduling.

**Multilevel Queue Scheduling** When process can be easily classified into different groups then Multilevel Queue Scheduling (MLQ) is applied. MLQ scheduling processes are classified into different groups. For eg., interactive processes (foreground) and batch processes (background) are two types of processes because of their different response time requirements, scheduling needs and priorities. A MLQ scheduling algorithm partitions the ready queue into separate queues. Processes are permanently assigned to each queue. This is done based upon the properties such as memory size or process type. Each queue has its own scheduling algorithm. **For Example,** the interactive queue might be used Round Robin Algorithm. For scheduling while batch queue follows FCFS. A multilevel Queue with five queue is listed below according to order of priority.



**Multiple Queue Scheduling**

Now, there are different ways of managing queues.

**Possibility I:**  We can assign a time slice to each queue which it can schedule among different processes in its queue. Interactive (or foreground) processes can be assigned 80% of CPU time whereas batch (or background) processes are given 20% of the CPU time.

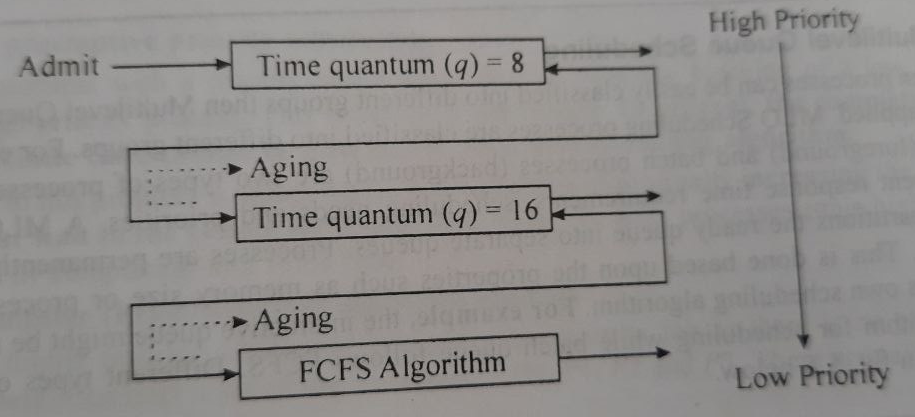
**Possibility II:** We can execute the high priority queue first. It means that no process in the batch queue could run unless the queue for system process & interactive process were all empty.

**Please note that if an interactive process entered the ready queue while a batch process is running, the batch process would be pre-empted.**

**Advantages of MLQ :-**  The processes are permanently assigned to a queue when they enter the system. Since processes do not change their foreground or background nature, so there is **low scheduling overhead.**

**Disadvantages of MLQ:- Starvation** of processes for CPU occurs here as the processes are never allowed to change their queues (inflexibility) and that if one or the other higher priority queues never become empty.

**Multilevel Feedback Queue Scheduling –** It is an enhancement of multilevel queue scheduling.  **The Principle here is that processes can move between the queue now.** In this scheduling, the ready queue is partitioned into multiple queues of different priorities. The processes are assigned to the queues by the system based on their **CPU burst characteristics.**  This means that if a process consumes too much of a CPU time, it is placed into a lower priority queue. So, I/O bound and interactive processes stay in the higher priority queues and CPU-bound processes move to the lower priority queues. However, these processes in lower priority queues should be promoted to the next higher priority queues. However, these processes in lower priority queues should be promoted to the next higher priority queue after a suitable time interval. **This technique of moving lower priority processes to the next higher priority queue is known as aging.**

****

**Multi-level Feedback Queues**

As shown in the figure above, the process queues are displayed from top to bottom in order to decreasing priority. The top priority queue has the smallest CPU time quantum when a process from the top queue exhausts this time quantum on the CPU then it is placed on the next lower queue. The queue’s time quota is scaled by a factor of 2 with each decreasing step in priority until the lowest priority queue is reached. A process that spends on amount of elapsed time exceeding the aging interval will be promoted to the next higher priority queue.

To implement this algorithm, we must know about the

1. Number of queues
2. Scheduling algorithm for each queue
3. Method of moving a process to a higher priority queue
4. Method of downgrading a process to a lower-priority queue
5. Method of assigning a process to a queue initially

**Advantages of MFQs**

1. It is a flexible scheduling algorithm as it allows the processes to move between the queues. So, I/O bound processes, do not have to wait long now.
2. Aging can be done. This form of aging prevents starvation.
3. It is the most general CPU scheduling algorithm and can be configured to match a specific system under design.

**Disadvantages of MFQs**

1. It is a complex scheduling algorithm.
2. The best scheduler is required.
3. Moving the processes around the queue produces more cpu overheads.

**Multiple processor scheduling**

multiprocessor scheduling focuses on designing the system's scheduling function, which consists of more than one processor. Multiple CPUs share the load (load sharing) in multiprocessor scheduling so that various processes run simultaneously. In general, multiprocessor scheduling is complex as compared to single processor scheduling. In the multiprocessor scheduling, there are many processors, and they are identical, and we can run any process at any time.

The multiple CPUs in the system are in close communication, which shares a common bus, memory, and other peripheral devices. So we can say that the system is tightly coupled. These systems are used when we want to process a bulk amount of data, and these systems are mainly used in satellite, weather forecasting, etc.

There are cases when the processors are identical, i.e., homogenous, in terms of their functionality in multiple-processor scheduling. We can use any processor available to run any process in the queue.

Multiprocessor systems may be ***heterogeneous*** (different kinds of CPUs) or *homogenous* (the same CPU). There may be special scheduling constraints, such as devices connected via a private bus to only one

There are two approaches to multiple processor scheduling in the operating system: Symmetric Multiprocessing and Asymmetric Multiprocessing.

1. **Symmetric Multiprocessing**: It is used where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.
2. **Asymmetric Multiprocessing**: It is used when all the scheduling decisions and I/O processing are handled by a single processor called the Master Server. The other processors execute only the user code. This is simple and reduces the need for data sharing, and this entire scenario is called Asymmetric Multiprocessing.

We have already studied the CPU scheduling algorithms for single CPU systems. What if there are multiple (many) CPUs? Herein, the scheduling problem is more complex, if we consider **Homogeneous systems** i.e.., systems with identical processors then there are certain issues concerning multiprocessor scheduling. Some of issues are given below –

1. With identical processor, **load-sharing** can occur.
2. With a separate queue for each processor, there could be situation where one CPU could be idle with an empty queue while other processor is very busy.

To avoid this situation, we could use a **common ready queue.** All processes enter one queue and are scheduled onto any available processor. In such a scheme, one of the two scheduling approaches may be used

**Case 1:** Each processor is **Self-scheduling.**  Each processor executes after examining the ready queue and selects a process to be executed.  **Please note that we have to take care so that no two processors should select the same process.**

**Case 2:**  One processor is appointed as the scheduler for the other processor, thus creating a master slave structure.

**Lottery Scheduling**

Lottery Scheduling is type of process scheduling, somewhat different from other Scheduling. Processes are scheduled in a random manner. Lottery scheduling can be preemptive or non-preemptive. It also solves the problem of starvation. Giving each process at least one lottery ticket guarantees that it has non-zero probability of being selected at each scheduling operation.

In this scheduling every process has some tickets and scheduler picks a random ticket and process having that ticket is the winner and it is executed for a time slice and then another ticket is picked by the scheduler. These tickets represent the share of processes. A process having a higher number of tickets give it more chance to get chosen for execution.

Underlying lottery scheduling is one very basic concept: tickets, which are used to represent the share of a resource that a process (or user or whatever) should receive. The percent of tickets that a process has represents its share of the system resource in question.

Let’s look at an example. Imagine two processes, A and B, and further that A has 75 tickets while B has only 25. Thus, what we would like is for A to receive 75% of the CPU and B the remaining 25%. Lottery scheduling achieves this probabilistically (but not deterministically) by holding a lottery every so often (say, every time slice). Holding a lottery is straightforward: the scheduler must know how many total

tickets there are (in our example, there are 100). The scheduler then picks a winning ticket, which is a number from 0 to 991. Assuming A holds tickets 0 through 74 and B 75 through 99, the winning ticket simply determines whether A or B runs. The scheduler then loads the state of that winning process and runs it.

Text

Description automatically generated with medium confidence

As you can see from the example, the use of randomness in lottery scheduling leads to a probabilistic correctness in meeting the desired proportion, but no guarantee. In our example above, B only gets to run 4 out of 20 time slices (20%), instead of the desired 25% allocation. However, the longer these two jobs compete, the more likely they are to achieve the desired percentages.

Text, letter

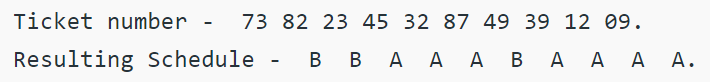
Description automatically generated

**Example** – If we have two processes A and B having 60 and 40 tickets respectively out of total 100 tickets. CPU share of A is 60% and that of B is 40%.These shares are calculated probabilistically and not deterministically.

**Explanation –**

1. We have two processes A and B. A has 60 tickets (ticket number 1 to 60) and B have 40 tickets (ticket no. 61 to 100).
2. Scheduler picks a random number from 1 to 100. If the picked no. is from 1 to 60 then A is executed otherwise B is executed.

**An example of 10 tickets picked by Scheduler may look like this –**

****

A is executed 7 times and B is executed 3 times. As you can see that A takes 70% of CPU and B takes 30% which is not the same as what we need as we need A to have 60% of CPU and B should have 40% of CPU. This happens because shares are calculated probabilistically but in a long run (i.e when no. of tickets picked is more than 100 or 1000) we can achieve a share percentage of approx. 60 and 40 for A and B respectively.

**Ways to manipulate Tickets-**

**Ticket Currency –**

Scheduler give a certain number of tickets to different users in a currency and users can give it to there processes in a different currency. E.g. Two users A and B are given 100 and 200 tickets respectively. User A is running two process and give 50 tickets to each in A’s own currency. B is running 1 process and gives it all 200 tickets in B’s currency. Now at the time of scheduling tickets of each process are converted into global currency i.e A’s process will have 50 tickets each and B’s process will have 200 tickets and scheduling is done on this basis.

**Transfer Tickets –**

A process can pass its tickets to another process.

**Ticket inflation –**

With this technique a process can temporarily raise or lower the number of tickets it own.

**Text & Reference books:**

1. Operating Systems: Three Easy Pieces, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci- Dusseau, Arpaci-Dusseau Books, May, (2014).