CHAPTER 3

Process Management

III.1. Introduction

In the execution of the different processes, it is necessary to have several machine resources contributing throughout the life cycle of a process, the most important is the resource that carries out the execution itself, it is the processor. CPU is the most important resource of any computing system as it is the only resource that executes instructions. Every process needs a CPU core non-shareable to execute its code.

During its life-time, a process uses CPU for execution and other resources (persistent memory and I/O devices) for other activities like read, write, display, print and so on. When the process uses other resources and does not require CPU, the CPU remains idle.

In a single process system, this is not a problem as the process can monopolize the CPU. But in a multiprogramming system where several processes are waiting to access a CPU core, keeping the CPU idle is a waste of time. To maximize the performance of a system, the CPU utilization should be maximum. In other words, whenever a CPU is free, we should allow other processes to use it. But at a single point in time, only one process can use the CPU. For the CPU, the operating system must provide a set of mechanisms to efficiently and optimally manage the processor's workload and control process execution.

This chapter describes the basic concept of process scheduling or CPU scheduling i.e. the task of an OS dealing with the allocation of a CPU or a CPU core to a set of processes.

III.2. Process Life Cycle

A process has a life cycle from birth to death, passing through various phases of running and waiting. The following figure illustrates the life cycle of a process in a single-processor system with a time-sharing strategy between processes.

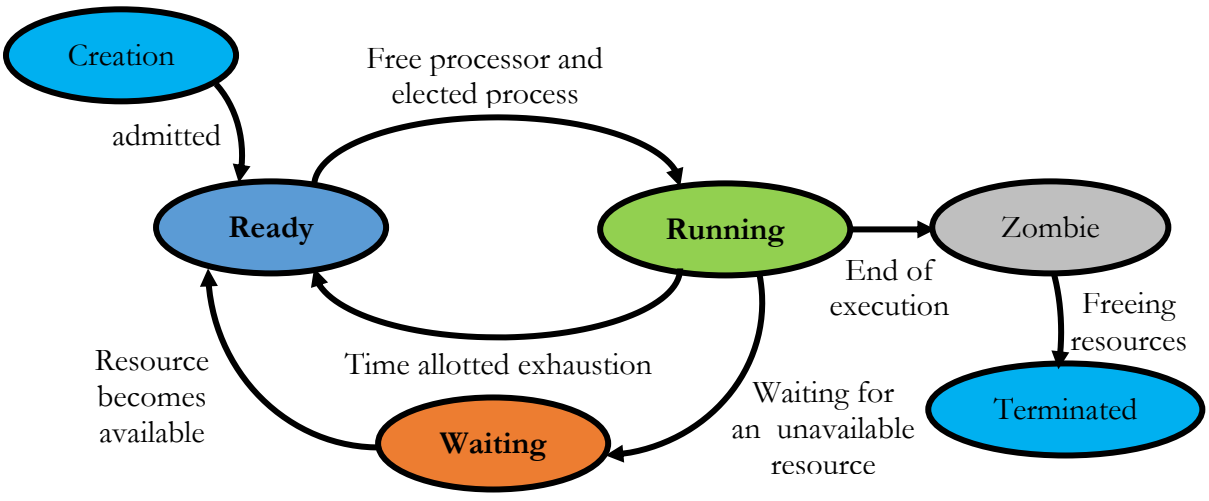


Figure 3.1 : Process Life Cycle

A process can be active if it is actually running, blocked if it is waiting for an event (end of I/O, delay, etc.), or ready if it is neither blocked nor active.

For each **ready and blocked state, a queue is associated with the processes** in that state, and managed by an OS tool.

III.3. Scheduling

Switching from the currently executing process to another process consists of two steps: selecting a process from among those that are eligible to use the processor, and giving control of the processor to the selected process. Software that implements the policy for selecting a process is called a **scheduler**; the algorithm it uses is called the **scheduling algorithm**.

The key to understanding a scheduler lies in knowing that a scheduler is merely a function. That is, the operating system scheduler is not an active agent that picks up the processor from one process and moves it to another. Instead, a running process invokes the scheduler: **A scheduler consists of a function that a running process calls to willingly give up the processor.**

Definition 3.1 (Scheduler)

The scheduling function manages the sharing of the processor between the different processes waiting to run, i.e. between the different processes that are in the ready state. The election operation consists in allocating the processor to a process.

Each time the processor becomes inactive, the operating system selects a process from the queue of ready processes, and passes control to it.

More concretely, this task is handled by two system routines: **Dispatcher** and **Scheduler**.

III.3.1. Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue. The Operating System maintains the following important process scheduling queues:

- **Job queue :** This queue keeps all the processes in the system. Processes in the job queue reside on mass storage and await the allocation of main memory.
- **Ready queue :** This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues :** The processes which are blocked (waiting) due to unavailability of an I/O device constitute this queue.

The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the below diagram, it has been merged with the CPU.

III.3.2. Types of Operating System Schedulers

The scheduler is an operating system module that selects the next jobs to be admitted into the system and the next process to run. Operating systems may feature up to three distinct scheduler types: a long-term scheduler (also known as an admission scheduler or high-level scheduler), a mid-term or medium-term scheduler, and a short-term scheduler. The names suggest the relative frequency with which their functions are performed.

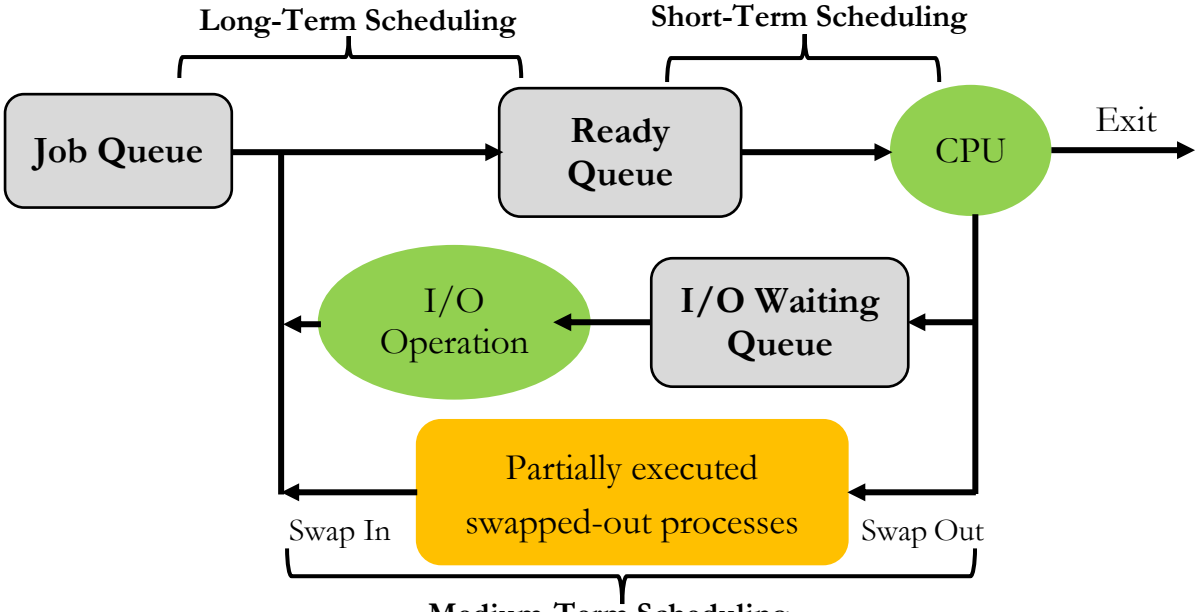


Figure 3.2 : Schedulers and Different Queues

III.3.2.1. Long-Term Scheduling

It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. On some systems, the long-term scheduler may not be available or minimal. Timesharing operating systems have no long-term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

III.3.2.2. Short-Term Scheduling

It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them. Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

III.3.2.3. Medium-Term Scheduling

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the

swapped out-processes. A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. **Swapping** may be necessary to improve the process mix.

III.3.2.4. Dispatcher

Another component that is involved in the CPU-scheduling function is the dispatcher, which is the module that gives control of the CPU to the process selected by the short-term scheduler. It receives control in kernel mode as the result of an interrupt or system call. The functions of a dispatcher mop the following:

- **Context switches**, in which the dispatcher saves the state (also known as context) of the process or thread that was previously running; the dispatcher then loads the initial or previously saved state of the new process.
- **Switching to user mode.**
- **Jumping** to the proper location in the user program to restart that program indicated by its new state.

The dispatcher should be as fast as possible, since it is invoked during every process switch. During the context switches, the processor is virtually idle for a fraction of time, thus unnecessary context switches should be avoided. The time it takes for the dispatcher to stop one process and start another is known as the **dispatch latency**.

III.3.2.5. Comparison among Scheduler

Long Term Scheduler	Short Term Schedular	Medium Term Scheduler
It is a job scheduler	It is a CPU scheduler	It is a process-swapping scheduler.
The slowest scheduler.	Speed is the fastest among all of them.	Speed lies in between both short and long-term schedulers.
It controls the degree of multiprogramming	It gives less control over how much multiprogramming is done.	It reduces the degree of multiprogramming.
It is almost absent or minimal in time sharing system	It is a minimal time-sharing system.	It is a component of systems for time sharing.
It can re-enter the process into memory, allowing for the continuation of execution.	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

III.4. Process Scheduling

CPU Scheduling is a kernel activity that involves context switch and change of states in the processes (from ready to running and running to ready or running to wait/blocked states). CPU scheduling is managed by an OS program - known as CPU scheduler. The scheduler runs in quick intervals, checks the queue of ready processes and allocates a CPU core to one process when the core is free. Therefore, the **process scheduler** is a part of the operating system that decides which

process runs at a certain point in time. It usually has the ability to pause a running process, move it to the back of the running queue and start a new process; such a scheduler is known as a **preemptive scheduler**, otherwise it is a **cooperative scheduler**.

III.4.1. Questions about Scheduling

- If there are **N** ready processes, which one should be allocated the CPU?
- Does a waiting process have to wait for the active process to finish?
- If an urgent (priority) process requests the CPU, does it have to wait?
- What are the CPU allocation policies (or strategies)?
- How can CPU utilization be optimized to maximize computer system performance?

III.4.2. When to Schedule

There are a variety of situations in which scheduling may occur. First, scheduling is absolutely required on two occasions:

- **When a process exits.**
- **When a process blocks on I/O, or a semaphore.**

In each of these cases the process that had most recently been running becomes unready, so another must be chosen to run next.

There are three other occasions when scheduling is usually done, although logically it is not absolutely necessary at these times:

- **When a new process is created :** It makes sense to reevaluate priorities at this time. In some cases, the parent may be able to request a different priority for its child.
- **When an I/O interrupt occurs :** This usually means that an I/O device has now completed its work. So, some process that was blocked waiting for I/O may now be ready to run.
- **When a clock interrupt occurs :** This is an opportunity to decide whether the currently running process has run too long.

III.4.3. Different Scheduling Policies

Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts: **preemptive** scheduling (with requisitioning) and **non-preemptive** scheduling (without requisitioning).

III.4.3.1. Non-Preemptive Scheduling

A non-preemptive scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) until it **voluntarily** releases the CPU. Obviously, non-preemptive algorithms have the potential problem of **monopolizing** the CPU, particularly by long processes when other processes suffer from indefinite block or **starvation**.

The starvation can lead to a catastrophe if the executing process goes into an infinite loop due to some programming errors. Preemptive algorithms do not suffer from this problem. But they cause frequent context switches and incur associated overhead.

III.4.3.2. Preemptive Scheduling

In contrast, a preemptive scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended (put in the ready queue) and the scheduler picks another process to run (if one is available).

Doing preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler. If no clock is available, non-preemptive scheduling is the only option.

III.4.4. Scheduling Criteria

Scheduling criteria are properties of entities, like jobs, processes, and threads, which can influence the order in which these entities are selected to use computer resources especially the CPU.

The aim of a CPU scheduler is to achieve "goodness" with respect to some measurable criteria. Different systems have different criteria, some are user-oriented and some system-oriented. Goodness according to each criterion is measured in terms of a performance metric. These criteria and a short description of each one are discussed below.

a) CPU Utilization: Utilization of any resource is defined as a ratio of its busy time and total time including its idle time. Hence,

$$\textbf{CPU Utilization} = \frac{\boldsymbol{CPU\ Busy\ Time}}{\boldsymbol{CPU\ Total\ Time}} = \frac{\boldsymbol{CPU\ Busy\ Time}}{\boldsymbol{CPU\ Busy\ Time + CPU\ Idle\ Time}}$$

- The utilization can be expressed as a fraction or in percentage. It should lie between **0** to **1** (or **0%** to **100%**). Higher its value, the better is the performance of the entire system as it means the CPU has lower idle time.
- Use **top command** in Mac OS and UNIX based systems to see CPU utilization. Its value **40%** or less means lightly loaded system, **90%** or more means highly loaded system.

b) Throughput: It is used to measure the performance of any system in terms of units of work or task accomplished in unit time.

- In case of CPU scheduling, it is defined as the number of processes completed in unit time (say in 1 second). Its value can be any positive real number per unit time.
- For a set of **long processes**, throughput can be a fractional value (say, **0.05 per sec**), whereas for very **short processes**, we can have integers (say, **10 per sec**).

c) Turnaround Time: It is the total time since a process is created to the time of its completion. Hence, it is the sum of wait time in the ready queue, CPU execution time, wait time in the I/O queue and time for doing I/O. Mathematically,

$$\textbf{TA Time} = \boldsymbol{TWR + TE + TW(I/O)\ + TD(I/O)}$$

TA Time: Turnaround Time

TWR : Total Wait Time in Ready Queue

TE : Total Execution Time in CPU

TW (I/O) : Total Wait Time in I/O Queue

TD (I/O) : Time for Doing I/O

- For any process, instruction execution in CPU and I/O activities are **not contiguous**. Both these activities rather happen in spells - few CPU bound instructions are followed by an I/O bound action and then again CPU bound instructions and so on.
- These spells are also called **bursts**.

The **average Turnaround Time** describes the average of the execution end dates:

$$\textbf{Avg TA Time} = \sum_{i=1}^{n} TA_i/n \ , \quad With \quad TA_i = End\ Time_i - Arrival\ Time_i$$

d) Burst Time: is defined as the time spent for executing the activity in the burst excluding the wait time in the queue. Hence, **Turnaround** (TA) time can also be defined as the sum of all CPU bursts and I/O bursts and wait-times in different queues.

$$\textbf{TA } Time = \sum CPU\ Bursts + \sum I/O\ Bursts + \sum Waiting\ Times$$

e) Waiting Time: is the time spent in the queue for the resource, from entry into the queue to use of the resource. In CPU scheduling, the waiting time corresponds to the time spent waiting in the ready CPU queue, unless otherwise specified.

$$\textbf{Avg WT} = \sum_{i=1}^{n} WT_i/n \ , \quad With \ \ WT_i = TA_i - Execution\ Time_i$$

f) Response Time: In interactive processes, users are more interested in getting the responses from the system. Users may tolerate delay in completion of the entire task if it may take time and continue in the background. In those cases, turnaround time is not that important, but what matters is the time spent in getting the first response from the system. Hence, response time is defined as the **time between submission of a request and getting the first response from the system** (not completion of production of the response).

For a good scheduling algorithm, we expect high CPU utilization, high throughput, low TA time, low wait time and low response time. However, not all such criteria can be met in a single algorithm. There are different algorithms to prioritize different criteria.

III.5. Scheduling Algorithms

CPU allocation or scheduling is to be done when there is at least one process in the ready queue and a CPU core is idle. For a single-process system, this is a trivial case and does not need any policy or algorithms. However, in a multiprogram environment (no matter whether a single core CPU or a multi-core one or multiple CPUs, number of processes are often way higher than the number of available cores), several processes contest to get a CPU core for executing instructions. We need a scheduling policy to determine which process will get the chance first and next, when and for how long. Scheduling algorithms implement one or more of such policies.

Let us focus on a few popular scheduling algorithms below.

III.5.1. First-Come-First-Served (FCFS) Algorithm

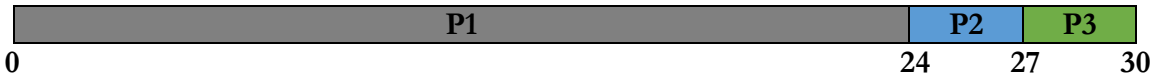
This is the simplest **non-preemptive** CPU scheduling algorithm. Every process is scheduled based on its **arrival time** (time of joining the ready queue of the CPU) and continues to run until it is

complete or voluntarily leaves CPU (I/O operation, Terminated, ...). When the CPU is free, the process that has the earliest arrival time is scheduled next. FCFS can be implemented using a FIFO (First In, First Out) ready queue.

Example 1:

Three processes **P1**, **P2** and **P3** arrive at the system **in this order (Arrival Time = 0).** Their execution times are respectively: **24**, **3**, and **3** time units **(T.U)**. To represent the processor occupancy history, we use the following diagram, known as a **Gantt chart**:

Gantt chart:



This diagram shows that process **P1** occupies the processor from time **0** to time **24**. At time **24**, the processor is occupied by process **P2**, followed at time **27** by process **P3**.

- **Waiting time** is **0** for process **P1**, **24** for process **P2** and **27** for process **P3**.
- The average waiting time is equal to : **(0 + 24 + 27)/3 = 17 (T.U)**.

Method Criticism

- Even though it is simple, it is not a very efficient algorithm as far as performance is concerned.
- The long processes can hold the CPU for long causing the short processes to wait for a long time (known as the **convoy effect**). This is the major weakness of the FCFS algorithm.
- The FCFS algorithm is particularly inconvenient for time-sharing systems.

III.5.2. Shortest-Job-First (SJF) Algorithm

It is a non-preemptive algorithm like the FCFS algorithm. Let's assume that we can know the execution time of each task in advance (In general, we will not know, so this is not meant as a practical policy! Rather, we use it as a thought experiment; later on). SJF runs the shortest job first, then the next shortest, and so on. If several processes have the same CPU burst time, a FIFO policy is used to decide between them.

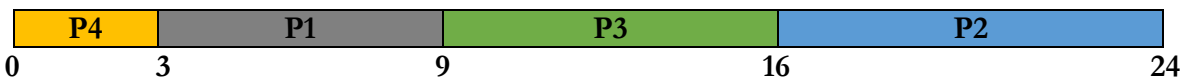
Example 2:

The system is subjected to four processes **P1**, **P2**, **P3** and **P4 in this order (Arrival Time = 0).** Whose execution times are given in the following table:

Process	Execution Time (T.U)
P1	6
P2	8
P3	7
P4	3

The Shortest-Job-First (SJF) algorithm will then give the following result:

Gantt chart:



- The **average waiting time** is = **(3 + 16 + 9 + 3)/4=7 (T.U)**.
- Whereas if we had chosen an FCFS policy, the average **waiting** time would be : **10.25 (T.U)**.

Method Criticism

- It has been proven that the SJF algorithm is **time-optimal** in the sense that it obtains the **shortest waiting time** for a given set of processes.
- However, this algorithm is difficult to implement for a simple reason: How can we know the execution time of a process in advance?

III.5.3. Shortest Remaining Time First (SRTF) Algorithm

A preemptive version of Shortest Job First (SJF) is Shortest Remaining Time First (SRTF). With this algorithm, the short-term scheduler always chooses the process that has term shortest remaining time. When a new process joins the ready queue, short term scheduler compare 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 connect to CPU. Otherwise continue the old process.

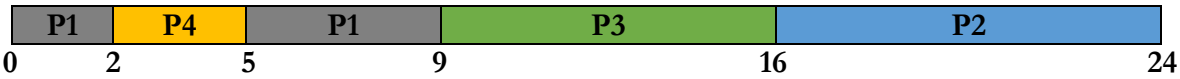
Example 3:

Let's consider the same problem as in **Example 2**. The four processes arrive at the system at different times, whose execution times and arrival times are shown in the following table:

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

The Shortest Remaining Time First (SRTF) algorithm will then give the following result:

Gantt chart:



- The **average waiting time** is = **(3 + 15 + 7 + 0)/4=6.25 (T.U)**.

Method Criticism

The disadvantages are also similar to the SJF algorithm. To be explicit, there are :

- The prior need for the execution time of the requests,
- The extra time it takes for the request queue to maintain its order based on the requests' remaining execution times,
- The priority of short requests over longer ones which constantly pushes back request, with long execution time, towards **starvation** when there are continuously arriving shorter requests,
- The overhead involved in keeping the remaining execution time of the running process (or running processes for multiprocessor systems) constantly updated and in **switching processes** whenever the remaining execution time of one of the processes in the ready queue is shorter than the remaining execution time of the running process.

III.5.4. Priority-based Scheduling Algorithm

The only criteria for scheduling processes may be priority. Priorities can be defined according to several parameters: process type, time limits, memory limits, etc. Also, priorities can be either **static**

(**fixed**), or **dynamic** (**variable**.) In the dynamic priority case, the priority of processes is periodically recalculated and updated.

Therefore, we can distinguish between two priority-based scheduling algorithms: **Priority Non-Preemptive Scheduling** and **Priority Preemptive Scheduling**.

a) Priority Non-Preemptive Scheduling

Generally, all processes are associated with a priority level. A scheduler allocates CPU to the process with the highest priority among all the processes in the ready queue. If two or more processes have the same priority level, they are then scheduled according to the FCFS principle.

SJF can be considered as a special case of the priority algorithm whereas if priority of a process is decided by the reciprocal of its CPU burst time (longest process is assigned the lowest priority).

b) Priority Preemptive Scheduling

Here, we considered a non-preemptive version of the priority algorithm. It suffers from the problem of indefinite blocking and causing starvation to other waiting processes. Hence, preemptive priority-based algorithms whenever a higher priority process arrives, the current process is preempted and the high priority one is scheduled immediately.

In other words, the operating system assigns a fixed priority rank to every process, and the scheduler arranges the processes in the ready queue in order of their priority. Lower-priority processes get interrupted by incoming higher-priority processes.

Example 4:

a) We have five processes with different priorities **(Arrival Time = 0)**, as shown in this table:

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

The **Priority Non-Preemptive Scheduling** algorithm will then give the following result:

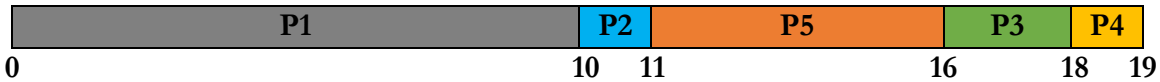
Gantt chart:

- The **average waiting time** is = **(6 + 0 + 16 + 18 + 1)/5 = 8.2 (T.U)**.

b) In this part, we add the arrival time of each process, as shown in this table:

Process	Execution Time	Priority	Arrival Time
P1	10	2	0
P2	1	4	1
P3	2	2	2
P4	1	1	3
P5	5	3	4

The **Priority Non-Preemptive Scheduling** algorithm will then give the following result:



- The **average waiting time** is = **(0 + 9 + 14 + 15 + 7)/5 = 9 (T.U)**.

The **Priority Preemptive Scheduling** algorithm will then give the following result:

Gantt chart:

- The **average waiting time** is = **(8 + 0 + 7 + 15 + 0)/5 = 6 (T.U)**.

Method Criticism

- Waiting time and response time depend on the priority of the process. Higher-priority processes have smaller waiting and response times.
- Deadlines can be met by giving processes with deadlines a higher priority.
- A deadlock situation can arise if low-priority processes wait indefinitely for the processor, while high-priority processes continue to stream in.
- To avoid such a situation, the so-called **aging technique** can be used. This involves gradually incrementing the priority of processes waiting in the system for a long time.
- **Starvation** of lower-priority processes is possible with large numbers of high-priority processes queuing for CPU time.

III.5.5. Round-Robin (RR) Algorithm

Round Robin (RR) is a **preemptive** scheduling algorithm based on First-Come, First-Served (FCFS) and is specifically designed for time-sharing systems. It is one of the oldest, simplest, and most widely used algorithms, known for its fairness and ability to prevent starvation.

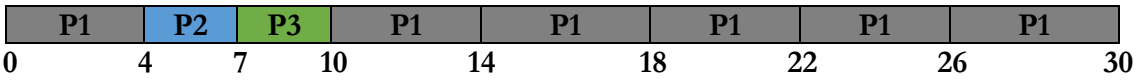
The basic idea of RR algorithm is simple:

- A **time quantum** (or **time slice**) is defined and a real-time clock (i.e., interval timer) is set to send an interrupt whenever the time quantum has passed, since the timer is set or reset. In order to repeat this cycle, the interval timer is reset again right after the interrupt is generated.
- Whenever the processor is ready to pick a new process for execution, it picks the process from the front of the ready queue. This can run for maximum of one-time quantum and the real time clock will send an interrupt as soon as the time quantum has passed.
- In response to this interrupt, the operating system stops the execution of this process and inserts it at the end of the ready queue. The scheduler then picks another process from the front of the ready queue.
- While a process is running it may want to do an I/O operation, may need a device which is not available, or may like to wait for an event to occur. The operating system will evacuate such a process from the CPU even if its time quantum has not completed and a new process will be picked up for immediate execution. To start a new time quantum for this process, the interval timer will be reset right before the new process is picked up.

Example 5:

Three processes **P1**, **P2** and **P3** arrive at the system **in this order (Arrival Time = 0).** Their execution times are respectively: **24**, **3**, and **3 ms**. Using a Round Robin (RR) algorithm, with a quantum of **4 ms**, we obtain the following Gantt chart:

Gantt chart:



- The **average waiting time** is = **(6 + 4 + 7)/3 = 5.66 ms**.

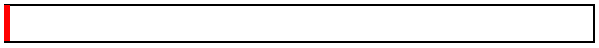
The last point about the RR scheduling strategy is the choice of time quantum length.

- An extremely long-time quantum will practically change the **RR** algorithm to a **FCFS** algorithm.
- A very short quantum, on the other hand, will reduce system efficiency (too frequent context switching).
- If the quantum is well chosen, the Round Robin method would enable processor sharing: each user would have the impression of having his own processor.
- The faster the processor, the shorter the time quantum is chosen.

Example 6:

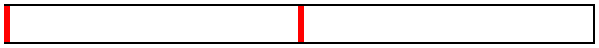
We have a process **P** with an execution time of **10 ms.** Let's calculate the number of context switches required for a quantum equal to : **12**, **6** and **1 ms** respectively.

Quantum = 12



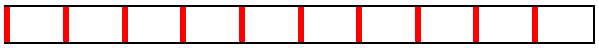
Number of context switches = 1

Quantum = 6



Number of context switches = 2

Quantum = 1



Number of context switches = 10

Method Criticism

- RR scheduling involves extensive overhead, especially with a small-time unit (context switching).
- Balanced throughput between FCFS/ FIFO and SJF/SRTF, shorter jobs are completed faster than in FIFO and longer processes are completed faster than in SJF.
- Good average response time, waiting time is dependent on number of processes, and not average process length.
- Because of high waiting times, deadlines are rarely met in a pure RR system.
- **Starvation can never occur,** since no priority is given.
- Order of time unit allocation is based upon process arrival time, similar to FIFO.

III.5.6. Multi-Level Queue Scheduling Algorithm

The Multilevel Queue Scheduling algorithm is implemented when processes in a ready queue are divided into different classes with different scheduling needs.

For example, we can consider a system with foreground and background processes. Here we can have two classes; one of the **foreground processes** and the other class of **background processes**; each having a separate set of scheduling needs. This situation is one where Multilevel Queue Scheduling can be implemented.

Let us consider an example of a multilevel queue-scheduling algorithm with four queues:

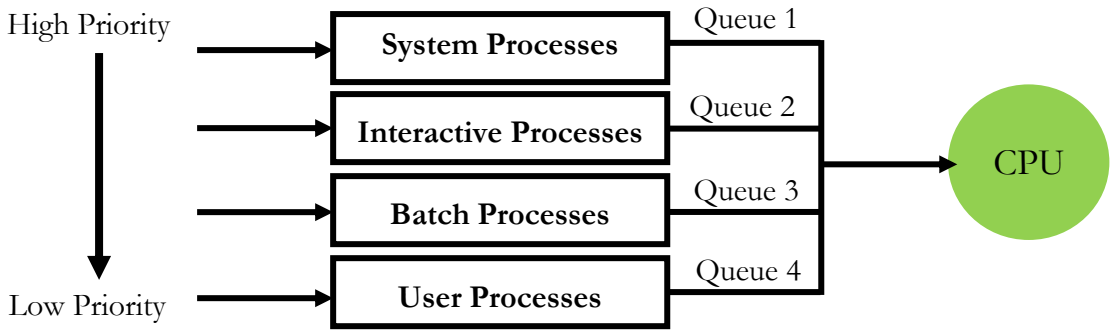


Figure 3.3 : Multi-Level Queue Scheduling

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 were all empty. If an interactive process entered the ready queue while a batch process was running, the batch process will be **preempted**.

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** is a process in which there should be the same kind of interaction (basically an online game).
- **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.
- **User Process:** The system process always gets the highest priority while the user 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.

Method Criticism

- The main disadvantage of Multilevel Queue Scheduling is the problem of **Starvation** for lower-level processes.
- It is not flexible in nature.
- These processes can't move across the queues.
- These disadvantages are overcome by multilevel feedback queue scheduling

III.5.7. Multi-Level Feedback Queue (MLFQ) Scheduling Algorithm

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. The **Ageing process** can be implemented in one of these ways. This form of **aging prevents starvation**.

In other words, the multilevel feedback algorithm selects always the first job of the lowest queue (i.e., the queue with the highest priority) that is not empty.

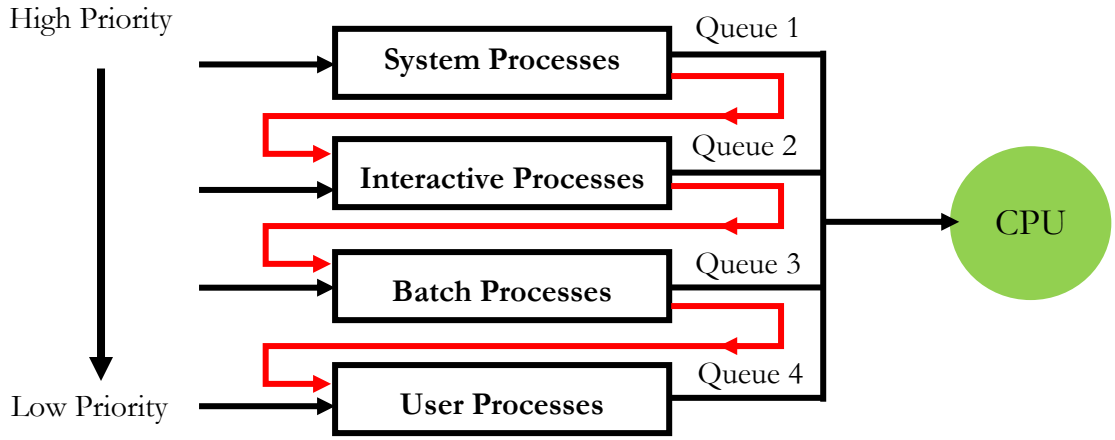


Figure 3.4 : MLFQ Scheduling Moved to a Lower-Priority Queue

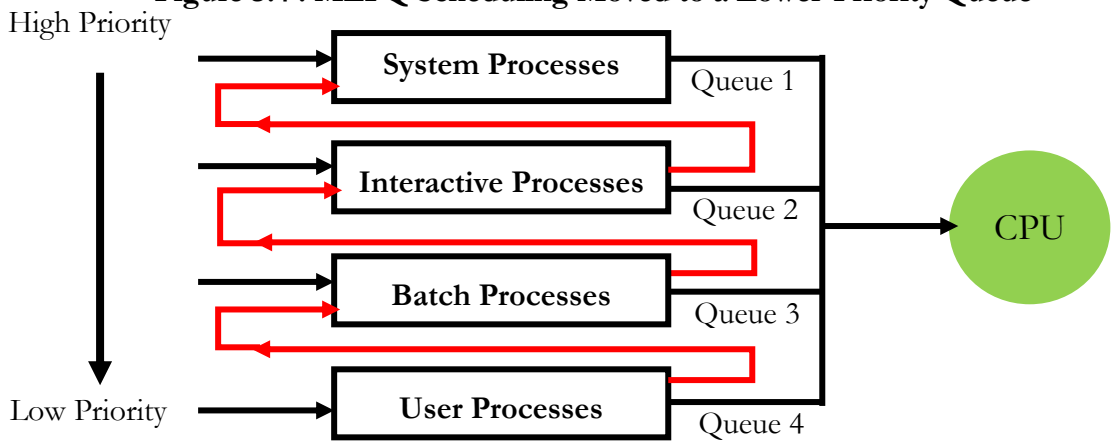


Figure 3.5 : MLFQ Scheduling Moved to a Higher-Priority Queue

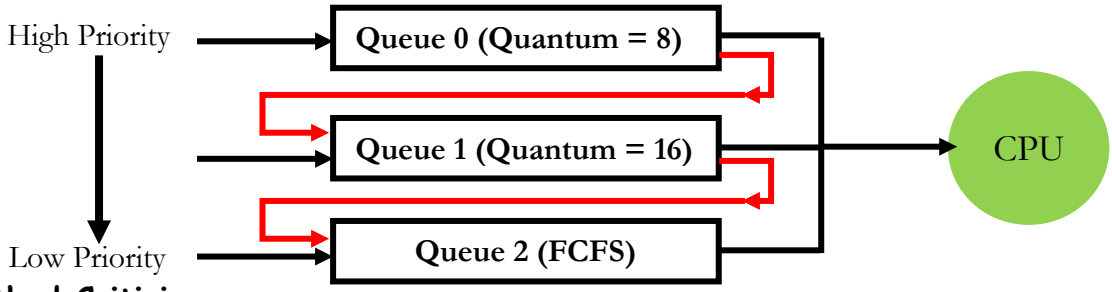
Example 7:

A system has three multi-level queues: **Queue 0**, **Queue 1** and **Queue 2**.

- **Queue 0** has the highest priority.
- **Queues 0** and **1** are managed using the **Round Robin** (RR) policy.
- **Queue 2** is managed according to **FCFS** policy.

A process entering the system will be placed in **Queue 0**. The process is given a time slot of **8 ms**. If it does not finish, it is moved to **Queue 1**.

- If **Queue 0** is empty, the process at the head of **Queue 1** is given a **16 ms** time slot.
- If it does not finish, it is interrupted and placed in **Queue 2**. Processes in **Queue 2** are executed only when **Queues 0** and **1** are empty.



Method Criticism

- Moving the process around queue produces more CPU overhead.
- The Multilevel Feedback Queue Scheduling algorithm is the most complicated algorithm
- Implementation is difficult.