**Chapter 06: CPU Scheduling**

Table of Contents

[CPU Scheduler 3](#_Toc83032977)

[Scheduling Criteria 3](#_Toc83032978)

[First-Come, First-Served Scheduling 5](#_Toc83032979)

[Shortest Job First Scheduling 7](#_Toc83032980)

[Non-Pre-Emptive 7](#_Toc83032981)

[Pre-Emptive 9](#_Toc83032982)

[Priority Scheduling 11](#_Toc83032983)

[Starvation 11](#_Toc83032984)

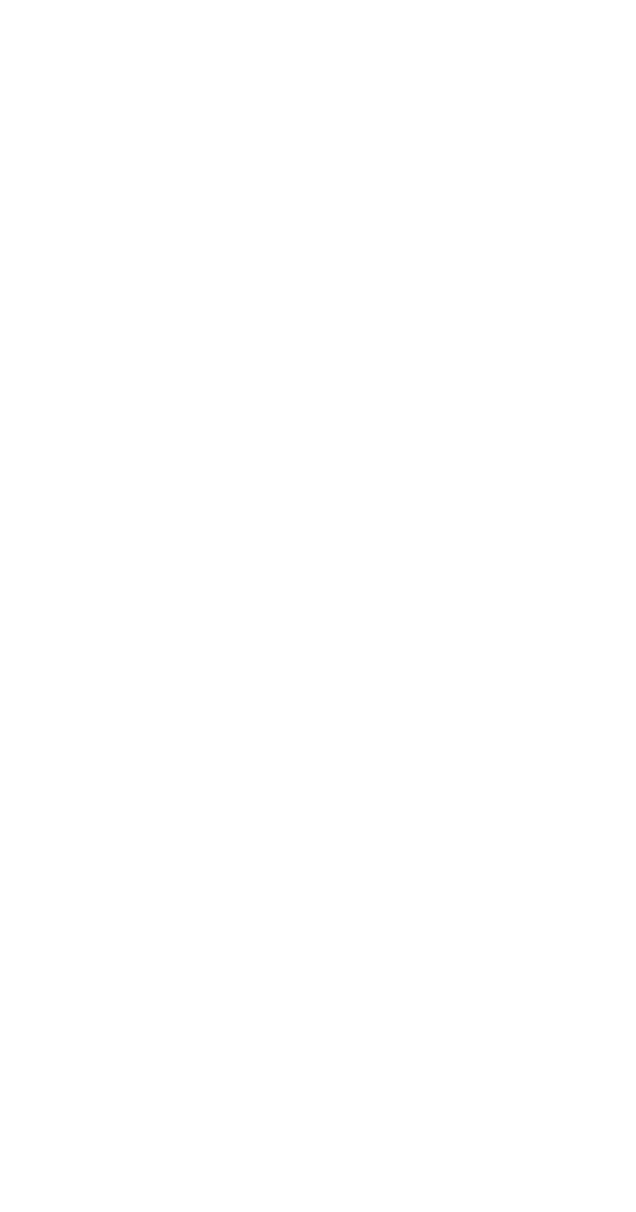
[Round Robin Scheduling 12](#_Toc83032985)

[Multilevel Queue 14](#_Toc83032986)

[Multilevel Feedback Queue Scheduling 15](#_Toc83032987)

**CPU Scheduling** is the basis for **multi-programmed** operating systems, which can run multiple processes in a single processor. Maximum CPU utilization is achieved through multiprogramming, since if the current process is not using the CPU, a different process is executed.

In a **continuous cycle**, a single process that is currently using the CPU may require an I/O device, for which it needs to wait. At that point, the CPU is taken away from that process and given to another. Process execution that consists of a cycle of CPU execution and I/O wait times is called **CPU-I/O Burst Cycles**.



## CPU Scheduler

The **CPU Scheduler** selects a process from the **ready queue** and allocates the CPU to it. There are also other queues involved, such as FIFO queues, priority queues, trees and unordered linked lists.

Scheduling decisions occur after specific events:

* A process switches from **running** to **waiting** state (I/O request).
* A process switches from **running** to **ready** state (interrupt).
* A process switches from **waiting** to **ready** state (I/O request satisfied).
* Process **termination**.

Among these events, scheduling that occurs due to the first and last ones are said to be **non-pre-emptive**, while scheduling that occurs due to the second and third are said to be **pre-emptive**. Non-pre-emptive scheduling cannot be interrupted, meaning they are atomic, while pre-emptive ones can. The meaning behind this will become clear once we look at the actual scheduling algorithms.

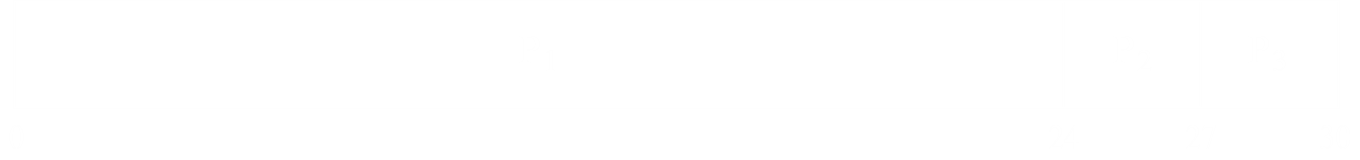
### Scheduling Criteria

* **CPU Utilization** – How busy the CPU is. We need to **maximize** this.
* **Throughput** – The number of processes that complete their execution per unit time. We need to **maximize** this.
* **Turnaround Time** – The amount of time required to execute a process, the time between when it is first arrives (enters the ready queue) to when it terminates, including any time spent waiting for I/O or waiting in the ready queue. We need to **minimize** this.
* **Waiting Time** – The amount of time a process spends waiting in the ready queue. We need to **minimize** this.
* **Response Time** – This is the amount of time between when a process arrives and when the first response is produced, i.e. the first time the process receives the CPU. We need to **minimize** this.

## First-Come, First-Served Scheduling

Suppose we have three processes, , and , with **burst times** , and respectively. The processes arrive in the order , and finally . All of them can be considered to have arrived almost together at time .

We can visualize the order in which the processes are executed using a **Gantt Chart**.



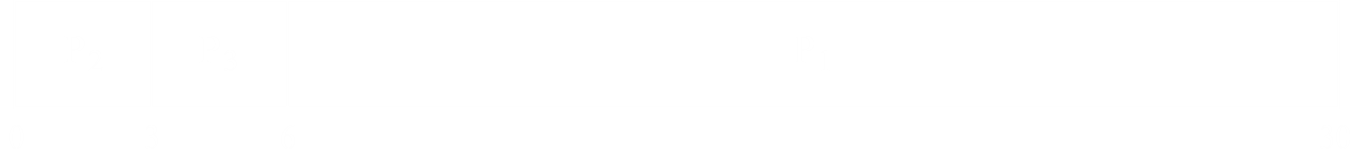
The **First-Come, First-Served** (FCFS) Scheduling mechanism is an example of **non-pre-emptive scheduling**. Even if arrives while is executing, we will not start executing until terminates.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Average |
| Turnaround Times |  |  |  |  |
| Waiting Times |  |  |  |  |
| Response Times |  |  |  |  |

Notice that the **Waiting Times** and **Response Times** are exactly the same. This is the case for all **non-pre-emptive** algorithms. For pre-emptive ones, we will later see that this will not be the case.

The table above was easy to fill up in this case, but for other algorithms, it is not as easy. We will require formulae, which we will be looking into.

Instead of doing this, if we consider that the processes are served in the order , and then , the table becomes like this:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Average |
| Turnaround Times |  |  |  |  |
| Waiting Times |  |  |  |  |
| Response Times |  |  |  |  |

The results are much better. This is due to the **Convoy Effect**, which occurs due to short processes being placed before longer ones. This is like if we decided to execute a CPU-bound process first before several I/O-bound ones, instead of the other way around. Obviously, this would be better.

Sadly, the second scenario is not guaranteed to occur. Because of this, FCFS scheduling is not used for time-sensitive scenarios.

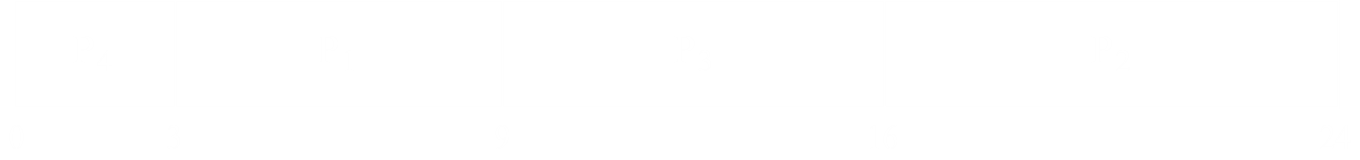
## Shortest Job First Scheduling

In **Shortest Job First** (SJF) Scheduling, we schedule processes based on their **burst time**. There are two schemes for this, non-pre-emptive and pre-emptive.

### Non-Pre-Emptive

The **non-pre-emptive** scheme works exactly like **FCFS**, except that we schedule the processes based on burst time instead of arrival time.

Suppose we have the process , , and , with burst times , , and respectively. All of them are considered to have arrived nearly together, at time . Thus, they are scheduled in the order , , and then .



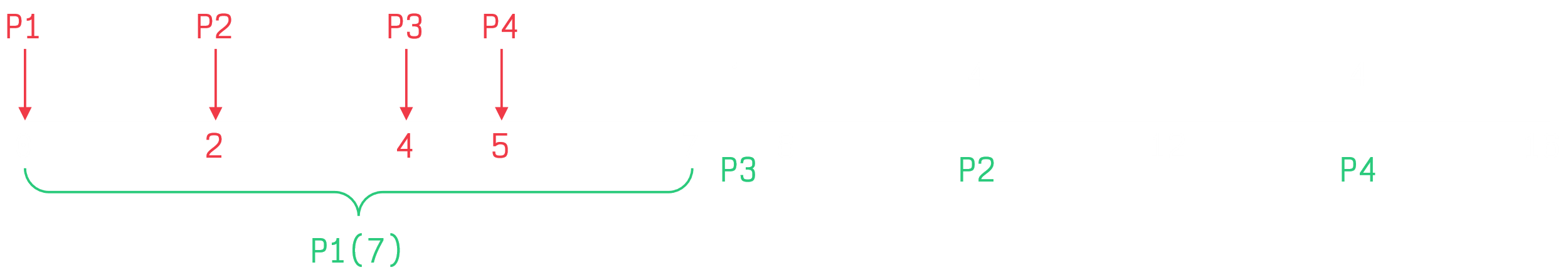
Notice that, after the initial sorting, the method is exactly the same as FCFS.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Average |
| Turnaround Times |  |  |  |  |  |
| Waiting Times |  |  |  |  |  |
| Response Times |  |  |  |  |  |

We will mostly be concerned about **Waiting Times**, so it makes sense to derive a formula at this point. Waiting Time , where is the time at which the process first gets the CPU and is the time at which the process first arrives. This seems simple for now, but as we move into other algorithms, we will see that there are other waiting times we need to consider.

Under **non-pre-emptive SJF**, say we have different **arrival times** now.

|  |  |  |
| --- | --- | --- |
|  | Arrival Times | Burst  Times |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |



starts at and ends at . At , we have , and waiting. Now we can just sort based on burst time, which means is executed first and then and are executed.

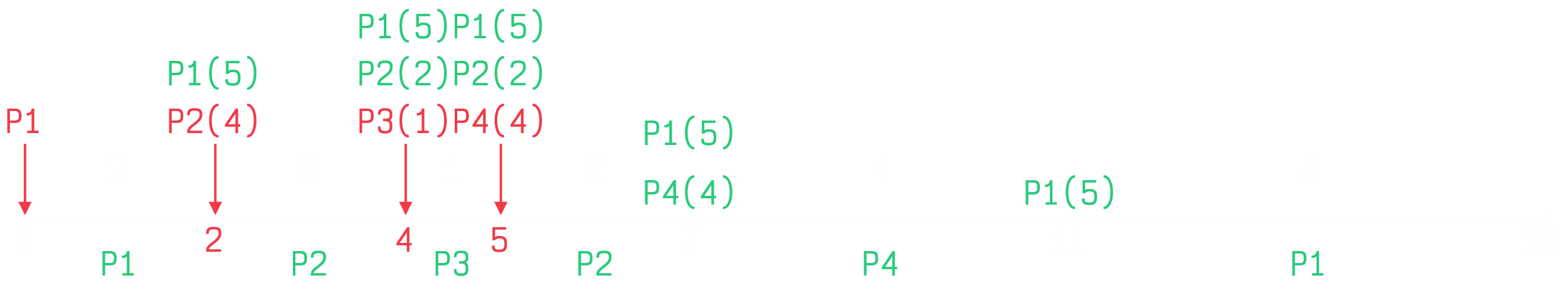
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | Average |
| Waiting Times |  |  |  |  |  |

### Pre-Emptive

Now consider a **pre-emptive SJF**, also called the **Shortest Response Time Function**, SRTF, of the same example as above.

|  |  |  |
| --- | --- | --- |
|  | Arrival Times | Burst  Times |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

This time, if a new process arrives, it will be given the CPU if it has a **shorter burst time** than the burst time remaining for the existing process.



At time , is executing.

At time , arrives. has a burst time of while the remaining burst time of is . Thus, starts executing.

At time , arrives. has a burst time of while the remaining burst time of is . Thus, starts executing.

At time , finishes execution. also arrives now with a burst time of . has the shortest remaining burst time, so it is executed next.

At time , finishes execution. has the shortest burst time, so it is executed next.

At time , finishes execution. is the only remaining process, so it starts execution.

In this case, calculating the **waiting time** is a little complicated, so we will use a formula.

|  |  |
| --- | --- |
|  | Waiting Time |
|  |  |
|  |  |
|  |  |
|  |  |
| Average |  |

Based on this, we can tell that the formula is:

where is the total waiting time, is the time at which the process first gets the CPU, is the arrival time, is the time at which the CPU is lost and is the time at which the CPU is achieved again.

The response time on the other hand, remains the same:

Thus, the **response time** and the **waiting time** for pre-emptive algorithms will not be the same.

## Priority Scheduling

In **Priority Scheduling**, every process is given a **priority number**. A lower value indicates higher priority.

In **pre-emptive** priority scheduling, we simply interrupt the currently running process if a process with higher priority arrives.

In **non-pre-emptive** priority scheduling, we execute the process with the highest priority, but without interrupting currently running processes.

SJF is also arguably a type of priority scheduling.

### Starvation

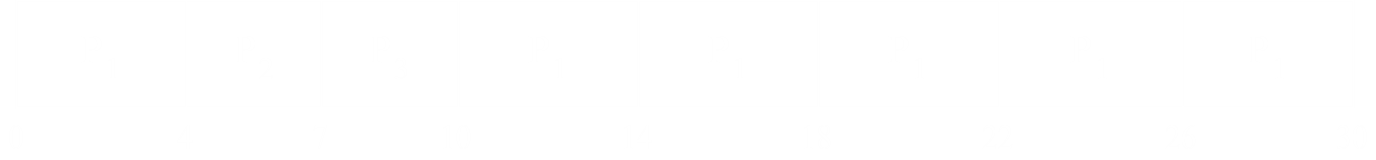
One major problem with priority scheduling is **starvation**. Low priority processes may never be executed. The solution to this is to use **aging**, where the priority of a process increases over time.

## Round Robin Scheduling

In the **Round Robin** algorithm, each process gets a small amount of CPU time, called the **quantum**. After this time has passed, the process is interrupted and placed at the end of the ready queue. Processes are not interrupted while they are being executed, so this algorithm is **non-pre-emptive**.

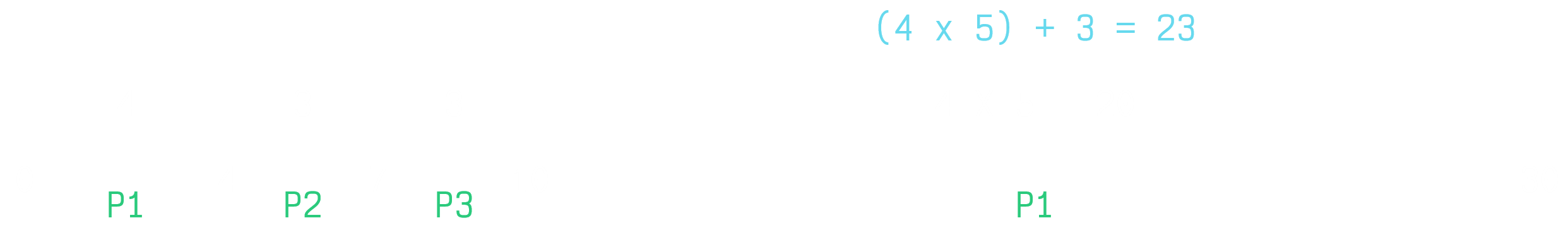
If we have processes and each process is allowed to execute for time units, a single process will end up waiting time units at most.

Say we have three processes, , and with burst times , and respectively and the quantum time is .



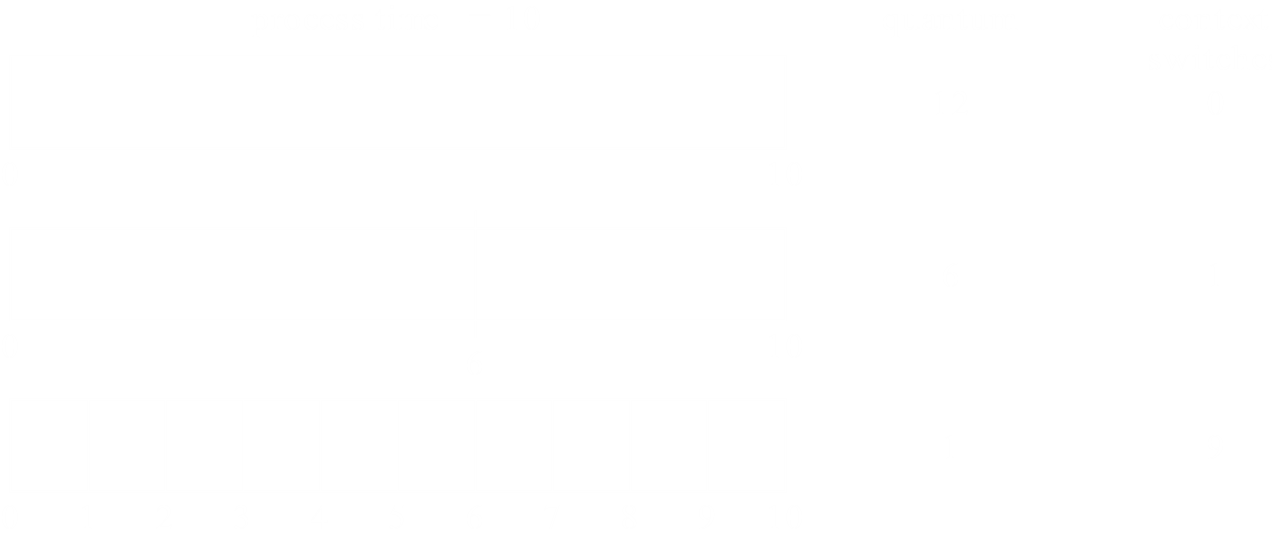
We could have given the remaining time unit after finishes execution to and then given the CPU to again at time , but that would involve more **context switching**.

Instead of dividing the blocks at the end into blocks, we could combine them. If we do, then the time should be shown as to indicate that the blocks should be combined. If the remaining burst time were , we would have written .

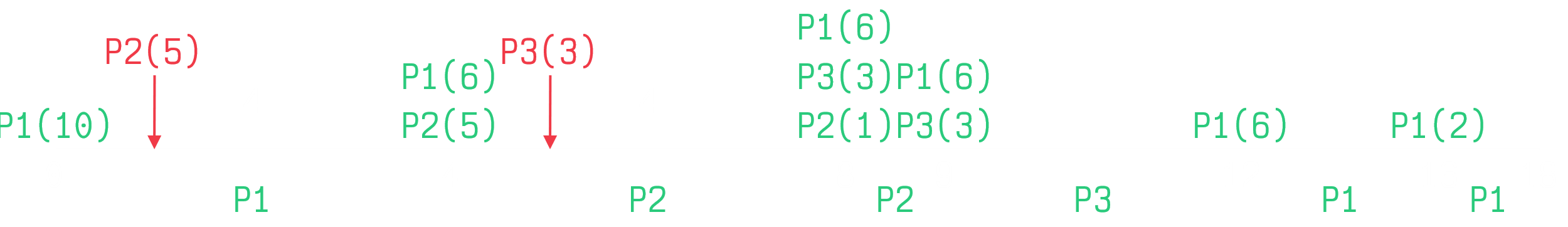


|  |  |
| --- | --- |
|  | Waiting Time |
|  |  |
|  |  |
|  |  |
| Average |  |

For a large value of , the algorithm behaves like a FIFO queue. For a small value of , the number of **context switches** increases, which increases overhead. Thus, we need to balance the value of .



A different interpretation of the Round Robin algorithm is to use **time blocks** but not use the round robin method. Thus, after a single quantum ends, the next process to be executed could be determined based on **priority** or on **remaining burst time**.



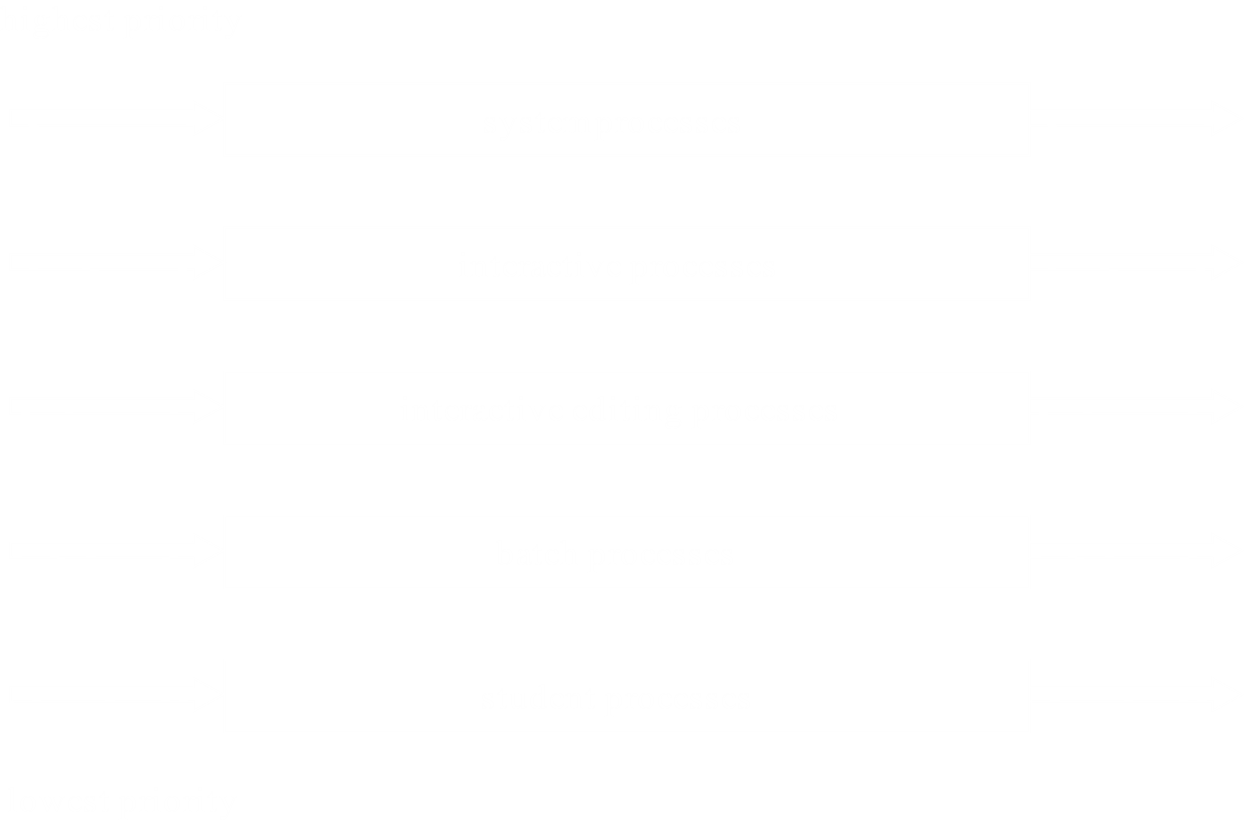
## Multilevel Queue

Processes can be of two types, **foreground processes**, which are interactive, and **background processes**, which are batch processes. These different process types have different response-time requirements and thus different scheduling requirements. Foreground processes may have priority over background ones.

To deal with the different scheduling requirements, we can divide processes into two or more **queues**, with each queue having its own scheduling algorithm. The foreground process queue is scheduled using the **Round Robin** algorithm while the background process queue is scheduled using the **First Come First Serve** algorithm.

However, we cannot execute multiple queues simultaneously, so we also need to figure out a way to schedule the queues themselves. This could be done in two ways:

* **Fixed Priority Pre-Emptive Scheduling** – Essentially, this means all foreground processes will be served before background ones. If we use this, it may cause starvation.
* **Time Slice** – Each queue gets a certain fixed amount of CPU time which it must schedule amongst its processes. For example, the foreground queue could get 80% of the time for Round Robin scheduling while the background queue gets 20% for First Come First Serve scheduling.



Note that processes **do not travel** between queues. System processes for example, will always stay in their own queue and not move to different ones.

### Multilevel Feedback Queue Scheduling

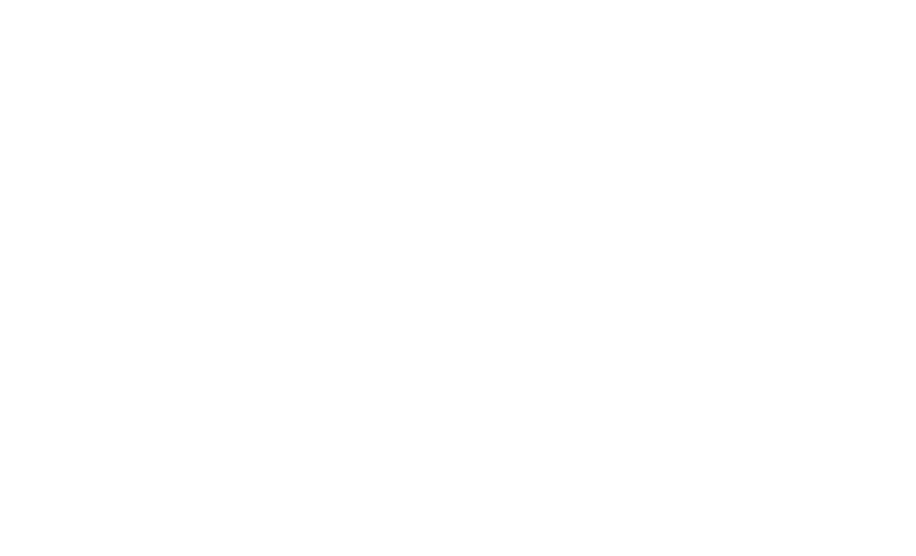
The last bit, not allowing processes to shift between queues, is a bit a problem. It can sometimes lead to **starvation**, with processes in lower priority queues not getting the opportunity to execute.

In a **Multilevel Feedback Queue Scheduler**, if a process is using **too much CPU time**, it is moved to a **lower priority queue**. If a process is **waiting too long** in a low-priority queue, it is moved to a **higher priority queue**.

A Multilevel Feedback Queue Scheduler depends on a few parameters:

* The number of queues.
* The scheduling algorithms for each queue.
* The method used to determine when to promote a process.
* The method used to determine when to demote a process.
* The method used to determine which queue a process must enter.

Consider that we have three queues, a queue using Round Robin scheduling with a quantum value of , another queue using Round Robin scheduling with a quantum value of and a third queue using First Come First Serve scheduling.



A new process would enter the first queue, where it would get 8 time units to execute. This is the lowest priority queue. If the process does not finish executing, then it is placed in the next queue.

In the second queue, it gets an additional 16 time units to finish executing. If it is still not done, it moves to the final queue.

The final queue has the highest priority. It is First Come First Serve, so once the process gets the CPU, it will not be interrupted until it is done executing.

Although the diagram does not show it, processes that have been waiting too long in the higher priority queues may be shifted to the lower priority ones as well, to avoid starving them.