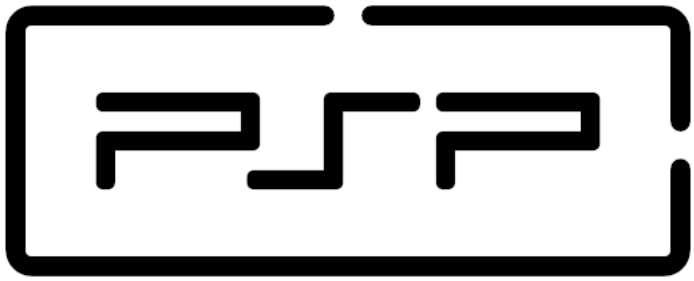


# Process and Service Programming

## 1.3. Processes in the OS

# 1.3. Processes in the Operating System



## 1.3.1. The OS kernel

The `kernel or OS core` is the responsible of the basic functions on the system and the resources management. It's accessed by systems calls. it is the smaller part of the OS and usually it's coded in low-level languages to improve its performance. The rest of the OS is call system apps.

Essentially, a process is what a program becomes when it is loaded into memory from a secondary storage medium like a hard disk drive or an removable drive. Each process has its own address space, which typically contains both program instructions and data. Despite the fact that an individual processor or processor core can only execute one program instruction at a time, a large number of processes can be executed over a relatively short period of time by briefly assigning each process to the processor in turn.

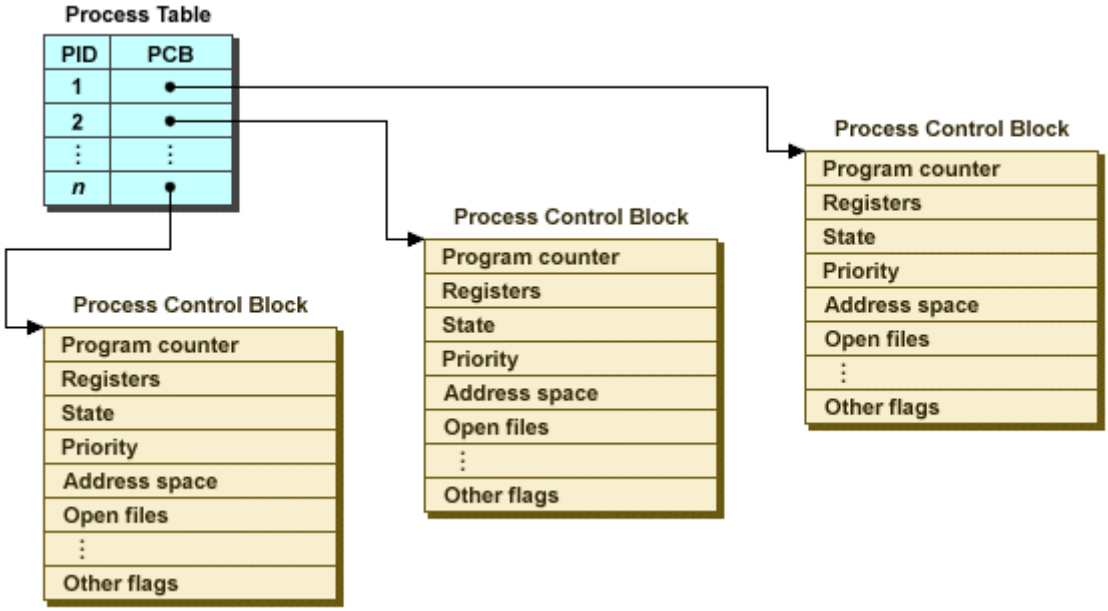
When a user starts an application program, the operating system's `high-level scheduler (HLS)` loads all or part of the program code from secondary storage into memory. It then creates a data structure in memory called a process control block (PCB) that will be used to hold information about the process, such as its current status and where in memory it is located.

The operating system also maintains a separate process table in memory that lists all the user processes currently loaded. When a new process is created, it is given a unique process identification number (PID) and a new record is created for it in the process table which includes the address of the process control block in memory.

As well as allocating memory space, loading the process, and creating the necessary data structures, the operating system must also allocate resources such as access to I/O devices and disk space if the process requires them. Information about the resources allocated to a process is also held within the process control block. The operating system's `low-level scheduler (LLS)` is **responsible for allocating CPU time to each process in turn**.

When a process makes the transition from one state to another, the operating system updates the information in its PCB. When the process is terminated, the operating system removes it from the process table and frees the memory and any other resources allocated to the process so that they become available to other processes. The diagram below illustrates the relationship between the process table and the various process control blocks.

These `context changes` are time and resource consuming. We will talk about this later, with a smaller running unit `threads`, that solve this problem partially.




The process control block (PCB) maintains information that the operating system needs in order to manage a process. PCBs typically include information such as the process ID, the current state of the process (e.g. running, ready, blocked, etc.), the number of the next program instruction to be executed, and the starting address of the process in memory. The PCB also stores the contents of various processor registers (the execution context), which are saved when a process leaves the running state and which are restored to the processor when the process returns to the running state.

## 1.3.2. Process conntrol in GNU/Linux

Because Linux is a multi-user system, meaning different users can be running various programs on the system, each running instance of a program must be identified uniquely by the kernel.

And a program is identified by its process ID (PID) as well as it's parent processes ID (PPID), therefore processes can further be categorized into:

- Parent processes – these are processes that create other processes during run-time.
- Child processes – these processes are created by other processes during run-time.

Init process is the mother (parent) of all processes on the system, it's the first program that is executed when the Linux system boots up; it manages all other processes on the system. It is started by the kernel itself, so in principle it does not have a parent process.

> **proceso init**
>
> The init process always has process ID of 1.
>
> It functions as an adoptive parent for all orphaned processes.

### Command to get the process PiD

In Linux every process on a system has a PID (Process Identification Number) which can be used to kill the process. The command `pidof cmdname` shows all processes related to that command. Remember that every time we start a command or application a nes process is created.

Shell variables `$$` and `$PPID` show the actual process PID and its PPID respectively.

```sh
# pidof systemd
1
# pidof top
2060
# pidof httpd
2103 2102 2101 2100 2099 1076
# Process pid
echo $$
2109
# Process parent pid
echo $PPID
2106
```

## Commands to view active processes in GNU/Linux

There are several Linux tools for viewing/listing running processes on the system, the two traditional and well known are `ps` and `top` commands:

The `ps` command displays information about a selection of the active processes on the system, along with some process information, as shown below:

This command offers many options to show more or less information about the processes, as well as our user's processes ot others' processes, including statistics about resource usage, etc.

```sh
vicente@Desktop-Vicente:~$ ps -AF
UID        PID  PPID  C    SZ   RSS PSR STIME TTY          TIME CMD
root         1     0  0   223   576   5 11:00 ?        00:00:00 /init
root         7     1  0   223    80   3 11:00 ?        00:00:00 /init
root         8     7  0   223    80   1 11:00 ?        00:00:00 /init
vicente      9     8  0  2508  5032   4 11:00 pts/0    00:00:00 -bash
vicente     70     9  0  2650  3224   5 11:06 pts/0    00:00:00 ps -AF
vicente@Desktop-Vicente:~$ ps -auxf
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0    892   576 ?        Sl   11:00   0:00 /init
root         7  0.0  0.0    892    80 ?        Ss   11:00   0:00 /init
root         8  0.0  0.0    892    80 ?        S    11:00   0:00  \_ /init
vicente      9  0.0  0.0  10032  5032 pts/0    Ss   11:00   0:00      \_ -bash
vicente     72  0.0  0.0  10832  3408 pts/0    R+   11:09   0:00          \_ ps -auxf
```

> **Useful 'ps' examples for Linux process monitoring**
>
> https://www.tecmint.com/ps-command-examples-for-linux-process-monitoring/ (https://www.tecmint.com/ps-command-examples-for-linux-process-monitoring/)

The `top` command is a powerful tool that offers you a dynamic real-time view of a running system as shown in the screenshot below:

```sh
vicente@Desktop-Vicente:~$ ps -AF
top - 11:14:52 up 14 min,  0 users,  load average: 0.00, 0.00, 0.00
Tasks:   5 total,   1 running,   4 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  12677.3 total,  12556.4 free,     70.6 used,     50.3 buff/cache
MiB Swap:   4096.0 total,   4096.0 free,      0.0 used.  12433.8 avail Mem
```

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    1 root      20   0     892    576    516 S   0.0   0.0   0:00.04 init
    7 root      20   0     892     80     20 S   0.0   0.0   0:00.00 init
    8 root      20   0     892     80     20 S   0.0   0.0   0:00.01 init
    9 vicente   20   0   10032   5032   3324 S   0.0   0.0   0:00.11 bash
   73 vicente   20   0   10856   3664   3148 R   0.0   0.0   0:00.00 top
```

> **'top' examples in Linux**
>
> https://www.tecmint.com/12-top-command-examples-in-linux/ (https://www.tecmint.com/12-top-command-examples-in-linux/)

## Process control

Process management is one of the important aspects of System Administration in Linux, and it includes killing of processes using the `kill` command.

When killing processes, the kill command is used to send a named signal to a named process or groups of processes. The default signal is the TERM signal.

A waiting process that can be interrupted by signals is called `Interruptible`, while a waiting process that is directly waiting on hardware conditions and cannot be interrupted under any conditions is called `uninterruptible`.

sh

```sh
# Get Firefox PID after it freezes
$ pidof firefox
2687
# Send the SIGKILL (9) signal to end the process immediately
$ kill 9 2687
```

> **How to control Linux process Using kill, pkill and kilall**
>
> https://www.tecmint.com/how-to-kill-a-process-in-linux/ (https://www.tecmint.com/how-to-kill-a-process-in-linux/)

The kernel stores a great deal of information about processes including process priority which is simply the scheduling priority attached to a process. Processes with a higher priority will be executed before those with a lower priority, while processes with the same priority are scheduled one after the next, repeatedly.

A user with *root* privileges can modify processes priority. This value can be seen in the NI (nice) columns of `top` output. This value also influences the PRI (priority) column, meaning the priority the OS gives to a process.

The priority is a nice value (niceness) which ranges from -20 (highest priority value) to 19 (lowest priority value) and the default is 0. Using the `nice` command we can guarantee that in high load CPU periods some processes will make a priority use of the CPU

sh

```sh
vicente@Desktop-Vicente:~$ nice
0
vicente@Desktop-Vicente:~$ nice -n 10 bash
vicente@Desktop-Vicente:~$ nice
10
vicente@Desktop-Vicente:~$
```

**Process control in Windows**

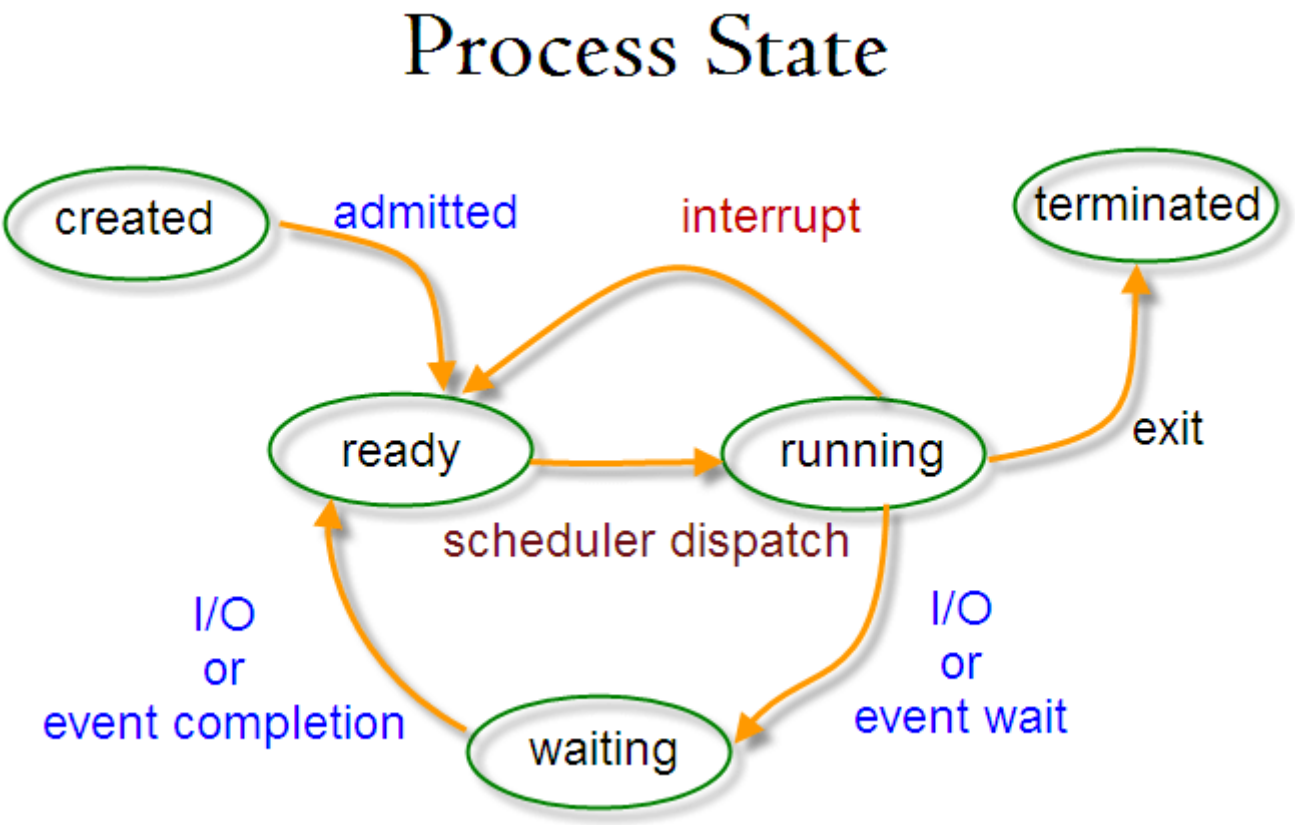
In Windows systems most of previous actions can be performed from the task manager, though commands **tasklist** and **taskkill** can be used in console mode..

*tasklist /svc /fi "imagename eq svchost.exe"* Will display you the result with Image name, PID and Service name to know which services are being run under the process svchost.exe, generic host services for services run from dynamic link libraries (DLL). There are so many processes for security reasons in order to avoid risks just in case one fails, not to hang the whole system.

## 1.3.3. Process states

The simple process state diagram below shows three main states for a process. They are shown as ready (the process is ready to execute when a processor becomes available), running (the process is currently being executed by a processor) and waiting (the process is waiting for a specific event to occur before it can proceed). The lines connecting the states represent possible transitions from one state to another.

At any instant, a process will exist in one of these three states. On a single-processor computer, only one process can be in the running state at any one time. The remaining processes will either be ready or blocked, and for each of these states there will be a queue of processes waiting for some event.




- **Created**. The process is created from a program and loaded into the system
- **Ready**. The process is not running but it is ready to do so. The OS still hasn't assigned a processor to run. and the OS scheduler will be responsible of selecting the process to start running.
- **Running**. While a process is executing it has complete control of the processor, but at some point the operating system needs to regain control, such as when it must assign the processor to the next process. Execution of a particular process will be suspended if that process requests an I/O operation, if an interrupt occurs, or if the process times out.

- **Waiting**. The process is blocked waiting for an event to happen. For instance it can be waiting for an I/O operation to finish or a synchronization operation with another process. When the event occurs the process goes back to ready state until the OS scheduler decides to move it to running state.
- **Terminated**. The process ends its processing and frees its resources and all memory space (PCB). The process is the responsible to do a system call to tell the OS it has finished although the OS can interrupt it forcing its termination by using an exception (special interruption).

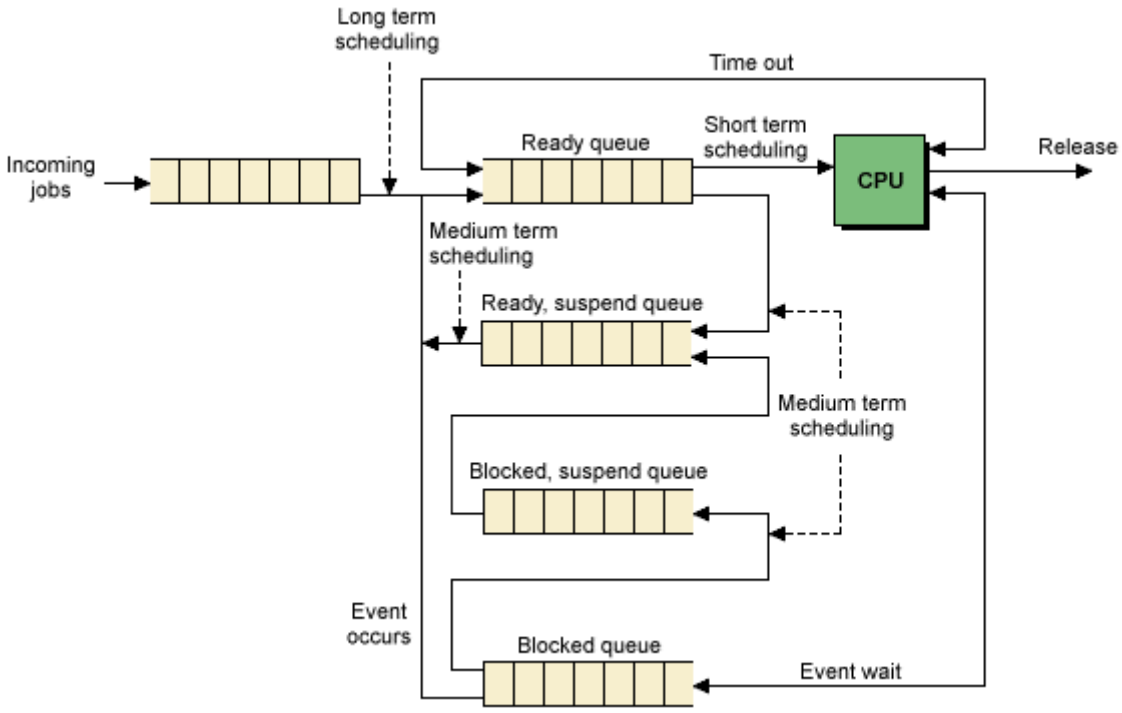
States transitions:

- **Running to waiting**: a process changes from running to waiting when it depends on an external event or operation.
- **De Waiting to ready**: a process changes from waiting to ready when the external event or operation it was waiting for occurs.
- **Ready to running**: a process changes from ready to running when the OS scheduler gives it CPU time.
- **Running to ready**: a process changes from running to ready when the CPU time given by the OS scheduler runs out.

## 1.3.4 Process scheduler

Process scheduling is a major element in process management, since the efficiency with which processes are assigned to the processor will affect the overall performance of the system. It is essentially a matter of managing queues, with the aim of minimizing delay while making the most effective use of the processor's time. The operating system carries out four types of process scheduling:

- Process queue: contains all system processes
- Ready queue: contains all processes ready to be run.
- Devices queues: contains processes waiting for an IO operation to finish.




Scheduler is the one who manages processes movements into the queues.There's a short-term and a long-term scheduling:

- The task of the `short-term scheduler` (sometimes referred to as the dispatcher) is to determine which process to execute next. This will occur each time the currently running process is halted. A process may cease execution because it requests an I/O operation, or because it times out, or because a hardware interrupt has occurred. The objectives of short-term scheduling are to ensure efficient utilization of the processor and to provide an acceptable response time to users.
  - Non-Preemptive Scheduling: a process only changes its state if it has finished or it gets locked.

- Preemptive Scheduling: a process only changes its state if it has finished, it gets locked or a higher priority process is waiting.
- Shared time: every amount of clock cicles (quantum), a process is moved to waiting and a new process changes from ready to running. All processes are considered to have the same priority

- The `long-term scheduler` determines which programs are admitted to the system for processing, and as such controls the degree of multiprogramming.
  - Before accepting a new program, the long-term scheduler must first decide whether the processor is able to cope effectively with another process. The more active processes there are, the smaller the percentage of the processor's time that can be allocated to each process.

> **Context switch**
>
> The changeover from one process to the next is called a **context switch**. During a context switch, the processor obviously cannot perform any useful computation, and because of the frequency with which context switches occur, operating systems must minimize the context-switching time in order to reduce system overhead.

## 1.3.5. Process scheduling algorithms

Scheduling algorithms are use to improve system performance and thus user experience.

To set objective parameters and be able to compare different scenarios, a CPU scheduling algorithm tries to maximize and minimize the following:

- **Waiting time**: Waiting time is an amount of time a process waits in the ready queue or in the waiting queue.
- **Turnaround Time**: Turnaround time is the amount of time to execute a specific process. It is the calculation of the total time spent waiting to get into the memory, waiting in the queue, locked for I/O operations and executing on the CPU. The period between the time of process submission to the completion time is the turnaround time.
- **CPU utilization**: CPU usage is the main task in which the operating system needs to make sure that CPU remains as busy as possible. It can range from 0 to 100 percent.

> In 1 processor systems
>
> $$\frac{\#\ instants\ of\ time\ the\ processor\ is\ busy}{time\ last\ process\ ends}\ x\ 100$$

> In N processor systems
>
> $$\frac{\sum_{n=1}^{N} \#\ instants\ of\ time\ the\ processor_n\ is\ busy}{instant\ of\ time\ when\ the\ last\ process\ ends\ *\ N}\ x\ 100$$

- **Throughput**: The number of processes that finish their execution per unit time is known as `throughput`. So, when the CPU is busy executing the process, at that time, work is being done, and the work completed per unit time is called throughput.

$$\frac{\#\ of\ processes}{instant\ of\ time\ when\ the\ last\ process\ ends}$$

| Process | Arrival | CPU time | Priority |
|---|---|---|---|
| P1 | 0 | 10 | 5 |
| P2 | 1 | 6 | 10 |
| P3 | 2 | 3 | 7 |

With this parameters let's compare the scheduling algorithms performance.

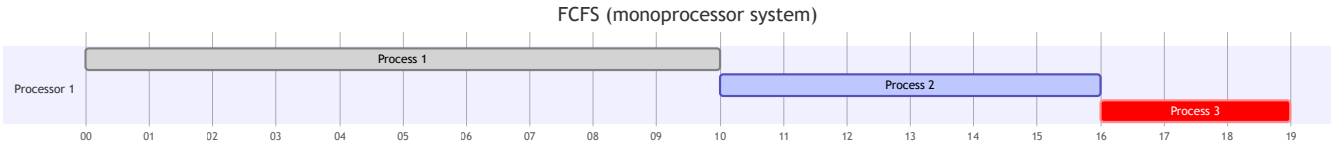
## FCFS - First Come First Served

First Come First Serve is the full form of FCFS. It is the easiest and most simple CPU scheduling algorithm. In this type of algorithm, the process which requests the CPU gets the CPU allocation first. This scheduling method can be managed with a FIFO queue.

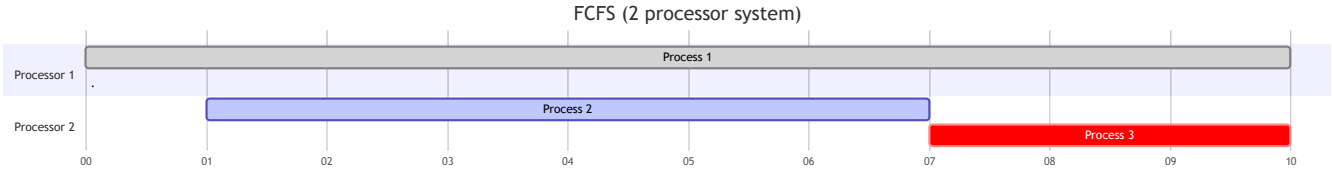
As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue. So, when CPU becomes free, it should be assigned to the process at the beginning of the queue.

Characteristics of FCFS method:

- It offers non-preemptive and pre-emptive scheduling algorithm.
- Jobs are always executed on a first-come, first-serve basis
- It is easy to implement and use.
- However, this method is poor in performance, and the general wait time is quite high.




| Process | Waiting time | Turnaround time | % CPU usage | Throughput |
|---|---|---|---|---|
| P1 | 0 | 10 | | |
| P2 | 9 | 15 | | |
| P3 | 14 | 17 | | |
| | | | | |
| Mean | 7,6 | 14 | 100% | 0,15 |



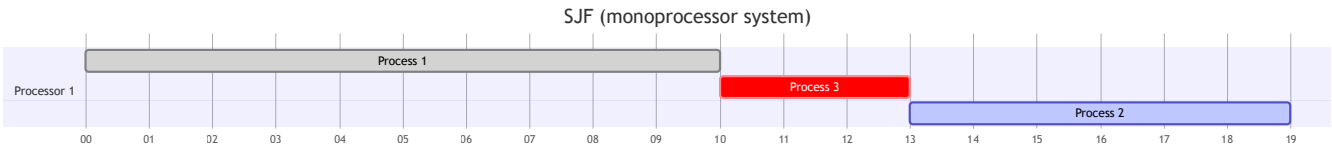

| Process | Waiting time | Turnaround time | % CPU usage | Throughput |
|---|---|---|---|---|
| P1 | 0 | 10 | | |
| P2 | 0 | 6 | | |
| P3 | 5 | 8 | | |
| | | | | |
| Mean | 1,6 | 6 | 95% | 0,3 |

## SJF - Shortest Job First

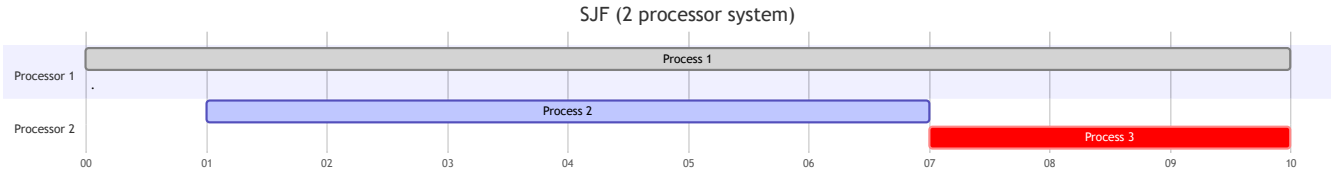
SJF is a full form of (Shortest job first) is a scheduling algorithm in which the process with the shortest execution time should be selected for execution next. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution.

Characteristics of SJF Scheduling

- It is associated with each job as a unit of time to complete.
- In this method, when the CPU is available, the next process or job with the shortest completion time will be executed first.
- It is Implemented with non-preemptive policy.
- This algorithm method is useful for batch-type processing, where waiting for jobs to complete is not critical.
- It improves job output by offering shorter jobs, which should be executed first, which mostly have a shorter turnaround time. There can be situations where longer jobs would never been run, that's called `starvation`.




| Process | Waiting time | Turnaround time | % CPU usage | Throughput |
|---|---|---|---|---|
| P1 | 0 | 10 | | |
| P2 | 12 | 18 | | |
| P3 | 8 | 11 | | |
| | | | | |
| Mean | 6,6 | 13 | 100% | 0,15 |



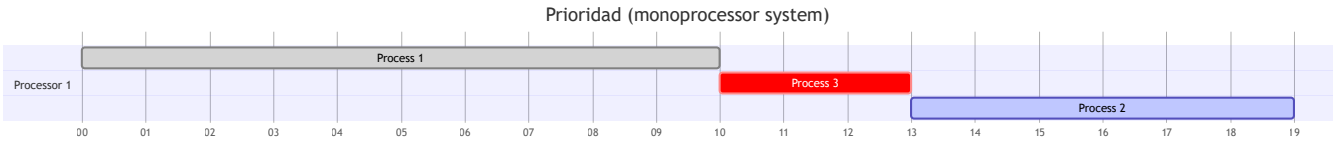

| Process | Waiting time | Turnaround time | % CPU usage | Throughput |
|---|---|---|---|---|
| P1 | 0 | 10 | | |
| P2 | 0 | 6 | | |
| P3 | 5 | 8 | | |
| | | | | |
| Mean | 1,6 | 6 | 95% | 0,3 |

## Priority scheduling

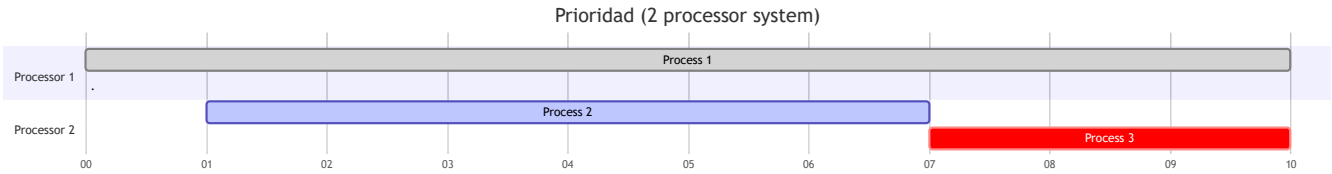
Priority scheduling is a method of scheduling processes based on priority. In this method, the scheduler selects the tasks to work as per the priority.

Priority scheduling also helps OS to involve priority assignments. 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 can be decided based on memory requirements, time requirements, etc.

As with SJF, with this algorithm low priority processes are in risk of starvation.




| Process | Waiting time | Turnaround time | % CPU usage | Throughput |
|---|---|---|---|---|
| P1 | 0 | 10 | | |
| P2 | 12 | 18 | | |
| P3 | 8 | 11 | | |
| | | | | |
| Mean | 6,6 | 13 | 100% | 0,15 |




| Process | Waiting time | Turnaround time | % CPU usage | Throughput |
|---|---|---|---|---|
| P1 | 0 | 10 | | |
| P2 | 0 | 6 | | |
| P3 | 5 | 8 | | |
| | | | | |
| Mean | 1,6 | 6 | 95% | 0,3 |

## Round Robin

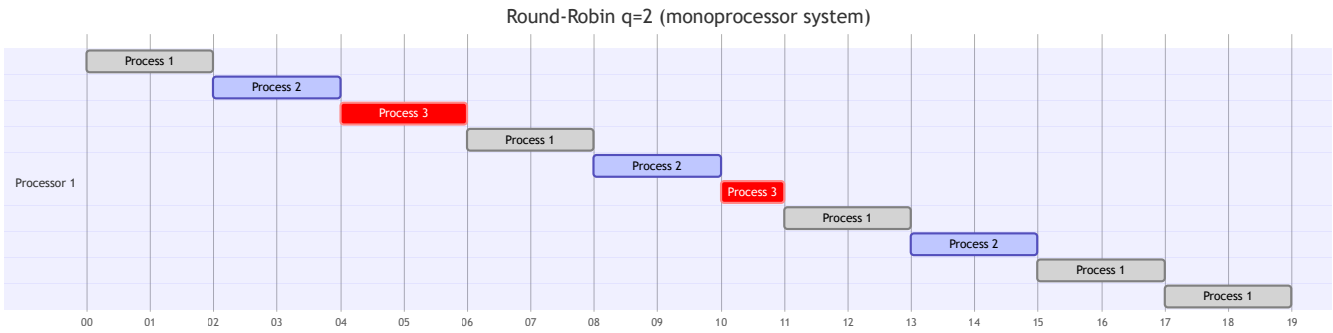
Round robin is the oldest, simplest scheduling algorithm. The name of this algorithm comes from the round-robin principle, where each person gets an equal share of something in turn ( `quantum` ). It is mostly used for scheduling algorithms in multitasking. This algorithm method helps for starvation free execution of processes.

Characteristics of Round-Robin Scheduling

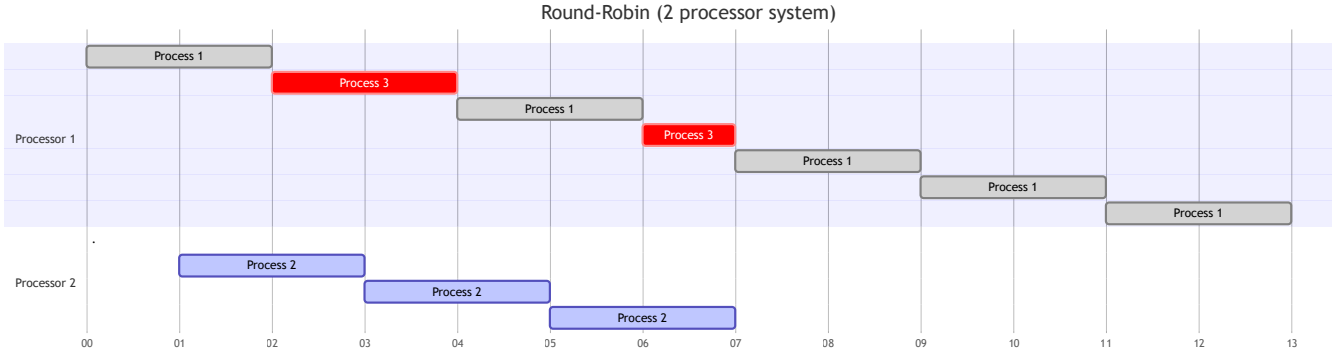
- Round robin is a hybrid model which is clock-driven
- Time slice should be minimum, which is assigned for a specific task to be processed. However, it may vary for different processes.
- It is a real time system which responds to the event within a specific time limit.

We can find two situations with this method:

- The process, or its remaining time, is less than the quantum. So, when the process finishes, a new process is run.
- The process, or its remaining time, is greater than the quantum. So, when the quantum times out the process is moved to ready and next scheduled process is moved to running.




| Process | Waiting time | Turnaround time | % CPU usage | Throughput |
|---|---|---|---|---|
| P1 | 9 | 19 | | |
| P2 | 8 | 14 | | |
| P3 | 6 | 9 | | |
| | | | | |
| Mean | 7,6 | 14 | 100% | 0,15 |




| Process | Waiting time | Turnaround time | % CPU usage | Throughput |
|---|---|---|---|---|
| P1 | 3 | 13 | | |
| P2 | 0 | 6 | | |
| P3 | 2 | 5 | | |
| | | | | |
| Mean | 1,6 | 7,6 | 73% | 0,23 |

**Combined scheduling**

As a matter of fact, not only one scheduling algorithm is used but more than one are combined to improve performance and avoid problems like starvation. We have done so, in Round-Robin we have also used FCFS.

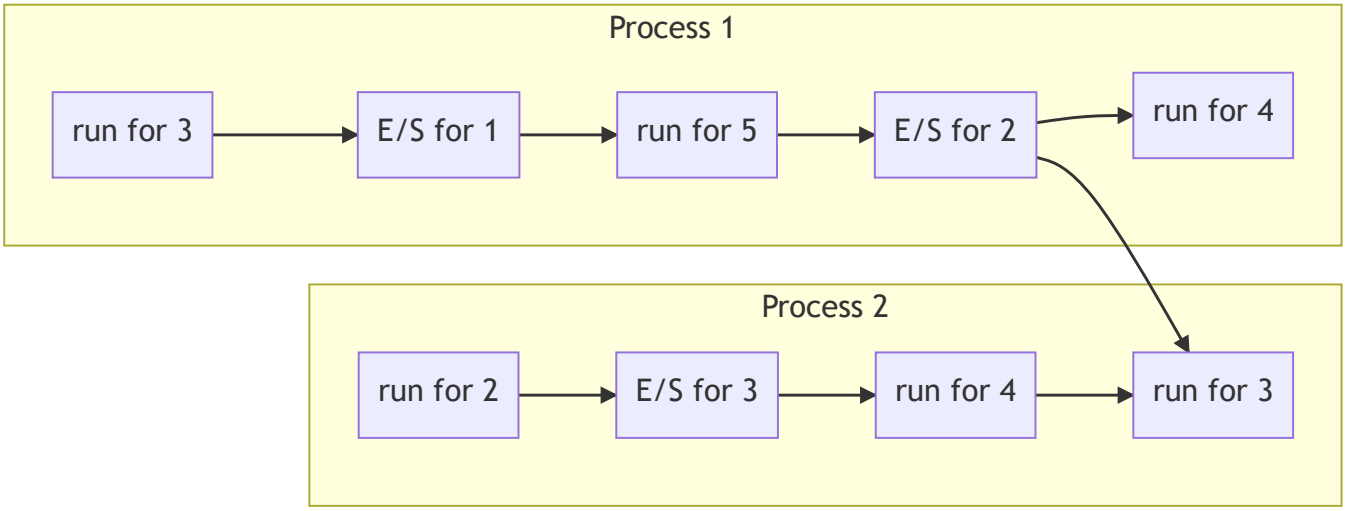
¿Do you dare to plan the previous sample using Round-Robin with priority? Keep in mind that it will work mainly with the quantum and, the priority will be used to select the next process to change from ready to running.

## Scheduler with I/O operations or locks

In previous examples all processes have expend all their time in CPU, the have not made any IO operation nor any interruption, but that behavior is far away from reality. Processes sometimes have to get locked to wait for a user input, read or store information on any storage or simply wait for another process to finish an operation and send to it a data before it can go on (synchronization).

As we have already comment, when a process leaves the running state another one can start running and make use of the CPU. Once the process finishes its lock, it can go back to ready state to keep on executing its sentences.

The next graph has a two activities specification in which before running their last sentence block, process 1 last IO operation must have finished.




Let's see how this affects the scheduling, guessing both processes arrive at the same time.