**PROCESS, THREADS & CPU SCHEDULING**

**Process**

* A process is not just a program code but also an **active entity**.
* **Active/dynamic entity:** Something that occupies **system resources** & occupation may vary time to time.
* **Passive/static entity:** Something that occupies system resources but occupation stays **constant until interrupted**.
* Attributes of process: **Hardware state, memory, CPU timing etc.**

OS process related goals:-

* **Share** the processor among various processes/tasks.
* **Provide** enough resources & response time to the processes.
* **Response time:** Answering a request made by a process.
* **Support** inter-process communication & synchronization.
* **Making sure** that user is able to create a process.

How are these achieved?

* By **scheduling & dispatching** processes to the processor.
* By **creating unbiased policies** for resource allocation to processes.
* By **maintaining internal database** of each current processes.

**Process Creation**

When is a process created?

* When a computer **initializes**.
* When the process used for creating process is called.
* When user **requests** to create a process.
* When a **batch of jobs** is submitted to CPU.

When is a process terminated?

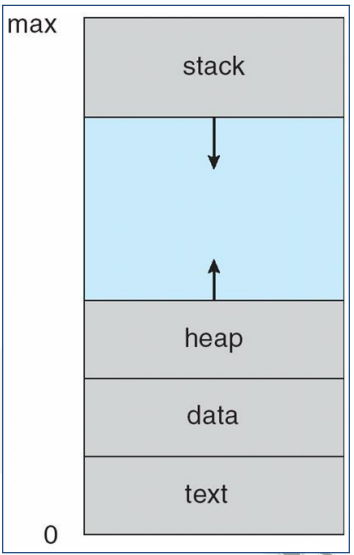
* **Volatile reasons:**
  + Normal exit
  + Error exit
* **Involatile reasons:**
  + Fatal error
  + Killed by another process

**Program Files**

* Programs are in general **executable (.exe)** files.
* These executable files are stored in **secondary memory**.
* When they are loaded into the **main memory**, it becomes a **process**.
* Process memory is divided into **4 parts**.

Process memory:-

* **Stack:** Contains the **local** variables in it & called **after** initialization.
* **Heap:** Contains the **dynamic** variables in it & called **after** initialization.
* **Text:** Contains the **compiled code** & called **during** execution.
* **Data:** Contains the **global/static** variables coded into the program & called **before** execution of main function.



**Process v/s Program**

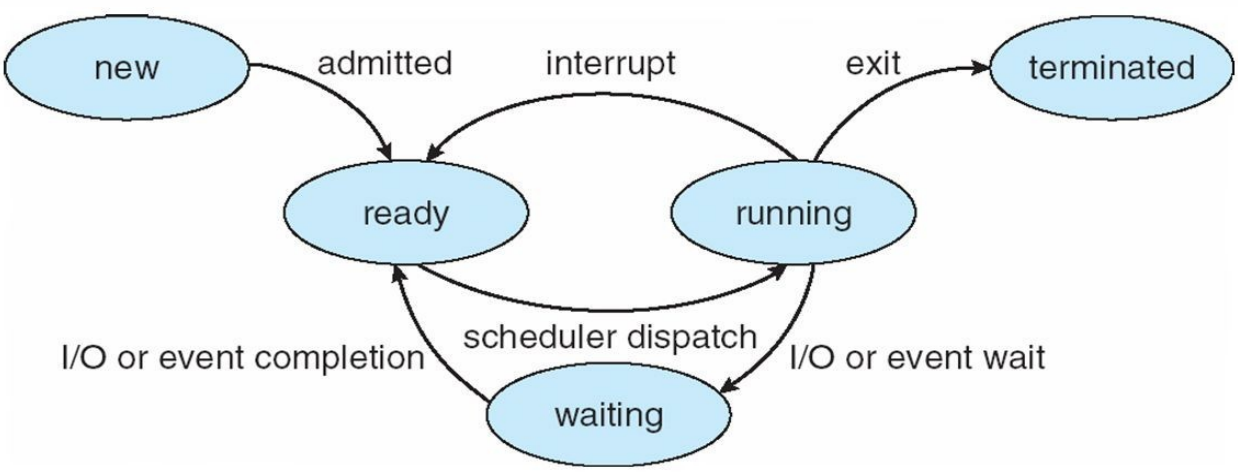
|  |  |
| --- | --- |
| **Process** | **Program** |
| **Program under execution.** | **Set of instructions.** |
| **Active/dynamic entity.** | **Passive/static entity.** |
| **Limited life span.** | **Comparatively longer life span.** |
| **May use some resources.** | **Just contains instructions in storage device.** |
| **Works in main memory.** | **Stored in secondary memory.** |
| **Occupies certain memory addresses called address space.** | **Requires space in secondary memory to be stored.** |

**Process State**

* A process **continuously** changes its state.
* **State:** Current activity levels or **nature** of the process.

Sates of a process:-

* **NEW** – The process is created.
* **READY** – The process is ready to be assigned to the processor.
* **OS scheduler**: Assigns process to processor.
* **RUNNING** – Program is under execution.
* **WAITING** – The process is under halt.
* **TERMINATED/EXIT** – Execution completed.



Process transition:-

* **READY -> RUNNING** – Dispatcher selects a new process to run.
* **RUNNING -> READY** – The file is interrupted by higher priority process.
* **RUNNING -> BLOCKED** – Process **can’t** go further due to unavailability of resources or waiting for another process to end.
* **BLOCKED -> READY** – When demands are met to it.

Suspend sub-processes:-

* **Blocked\_suspend** – The queue in which suspended processes which are **not** ready to run are stored.
* **Ready\_suspend** – The queue in which suspended processes which are now ready to run are stored.

**Process Control Block (PCB)**

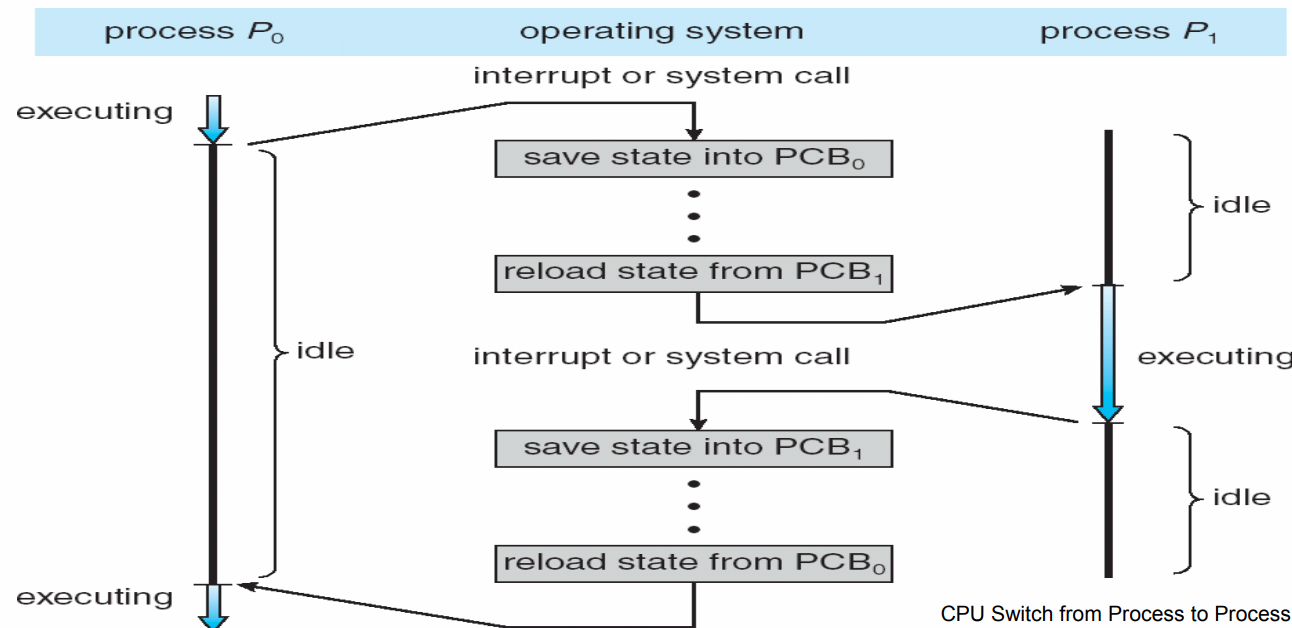
* Does **accounting** of each processes in an OS.
* Deleted after the process terminates.

This table includes:-

|  |  |
| --- | --- |
| **Attribute** | **Work** |
| **Process state** | **-** |
| **Process privileges** | **Tells what and what not process can access.** |
| **Process ID** | **-** |
| **Pointer** | **A pointer to parent process.** |
| **Program counter** | **Pointer to address of next instruction.** |
| **CPU registers** | **CPU registers where process will be stored.** |
| **CPU scheduling information** | **Process priority & other scheduling information.** |
| **Memory management information** | **-** |
| **Accounting information** | **Contains many precise figure data.** |
| **I/O status information** | **List of I/O devices involved in the process.** |

**Context Switching**

* State of one task is **saved** before switching to another.
* **Restoration:** Process rollback.



* It creates an **illusion** of tasks running altogether.
* Time taken when switching context is pure **overhead**.
* Context switching being a **bottleneck**, programmers now use threads.

**Threads**

* Also known as **lightweight** process.
* **Thread:** An execution unit which has its own **program counter**, **stack** & **registers**.
* Threads are executed **parallelly**.
* A process can have multiple threads, but one thread can be in one process only.

Advantages:-

* **Minimizes** context switching time.
* Concurrency
* More economical
* Proper **utilization** of microprocessor.

**Process v/s Thread**

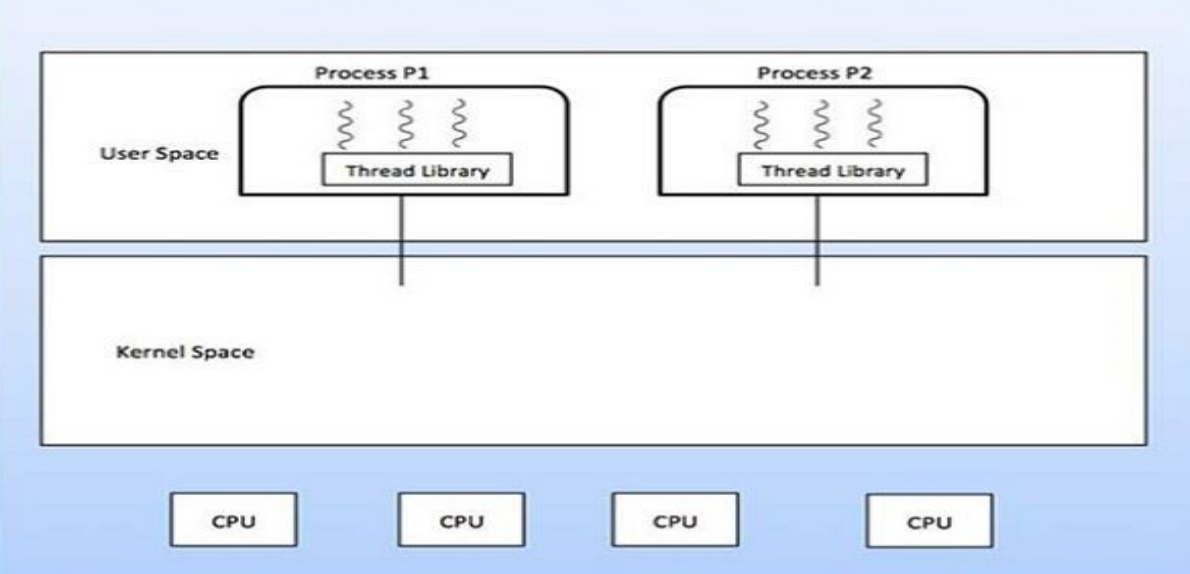
|  |  |
| --- | --- |
| **Process** | **Thread** |
| **Consumes more resources.** | **Consumes less resources.** |
| **Process switching requires interaction with OS.** | **Thread switching doesn’t require interaction with OS.** |
| **It executes same code but has its own resources.** | **Each thread gets even lesser amount of resources.** |
| **If one process blocks, others will run only when it unblocks and completes.** | **If one thread blocks, other concurrent thread will continue to work.** |
| **One process can’t interfere another.** | **A thread can interfere another thread.** |

**Types of Threads**

* User level threads
* Kernel level threads

**User level threads**

* Threads managed by **user**.
* Thread managing kernel **doesn’t** know about it.
* User uses **thread library** for programming.
* **Thread context:** Information about a thread.



Advantages:-

* Thread switching **doesn’t** depend on kernel.
* **Portable** OS to OS.
* Can be scheduled our **own way**.
* Operations on them are **fast**, despite being high-level.
* **Doesn’t** require multiprocessing system.

**Kernel level threads**

* Threads acting on **kernel**.
* Managed also by **kernel** & **OS**.
* **Can’t** be controlled by user or any application.

Advantages:-

* Kernel can schedule all threads to their slots, **all at once**.
* Kernal assigns a blocked thread’s slot to **another** thread.

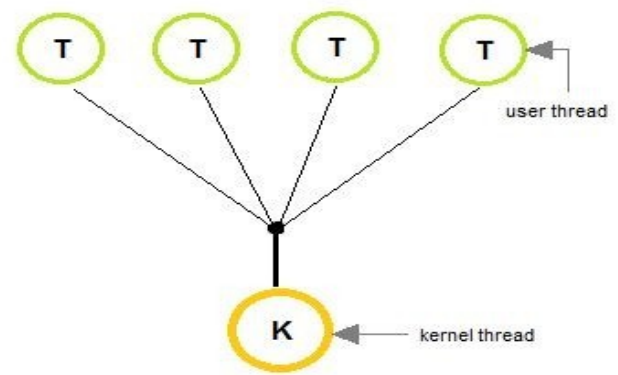
Disadvantages:-

* Operations on them are **slow**, despite being low-level.
* Kernel switches **mode** when control is switched one thread to another.

**Multithreading**

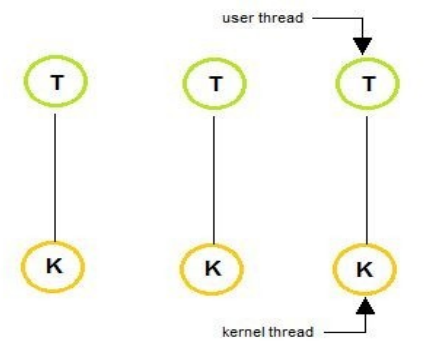
* Execution of multiple threads **at once**.
* **Multithreading modes:**
  + One-to-one
  + One-to-many
  + Many-to-one

**One-to-Many Multithreading**



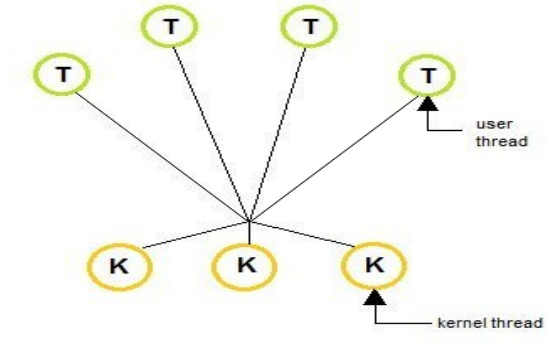
* Thread is managed by user with use of **programming libraries**.

**One-to-One Multithreading**



* **Linux** & **Windows 95** to **XP** used this.
* There is in general **limitation** to number of threads in such OS.

**Many-to-Many Multithreading**



* Combines **best features** of **one-to-one** & **many-to-one** models.
* **No limitation** on number of threads.
* Blocking kernel system call blocks a single thread, **not** entire process.
* Process is **split** among **multiple processors** in multi-processing OS.

**Benefits of Multithreading**

* Better utilization of resources.
* Economical
* Smooth context switching

**User-level Threads v/s Kernel-level Threads**

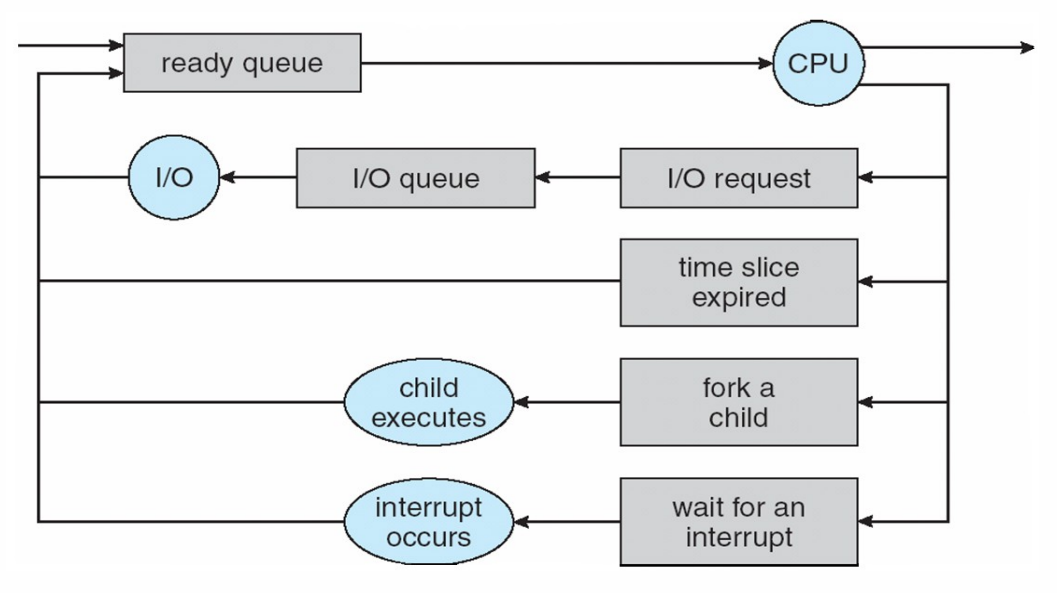
|  |  |
| --- | --- |
| **User Level Threads** | **Kernel Level Threads** |
| **Faster to create & manage.** | **Slower to create and manage.** |
| **Made using programming libraries.** | **Made using OS kernel.** |
| **Portable** | **Non-portable** |
| **Doesn’t use multiprocessing.** | **Uses multiprocessing.** |

**Process Scheduling**

* **Process scheduling:** Determining which processes to move from **job queue** to **processing queue**.
* Its purpose is to **keep CPU busy** & **reduce response time**.
* **Scheduler** applies some rules on processes which move processes from **CPU** to **memory** & vice versa.
* **Non pre-emptive scheduling:** One process **can’t** interfere another, **no** priority is given to any process by the CPU.
* In it, other jobs have to wait in **ready queue** before job currently executing completes.

**Scheduling Queue**

* When a process comes into the system, it is initially stored in the **job queue**.
* Processes waiting for a device to become available are stored in **device queues**.
* There are **unique device queues** for each peripheral device.
* **Dispatch:** Selection of a process for **execution**.



* Waiting queue comes **before** ready queue.
* When a process is terminated, it is removed from **all queues** and its PCB data is **deallocated**.
* Types of schedulers: **Long term**, **short term** & **medium term**.

**Types of Schedulers**

Long term scheduler:-

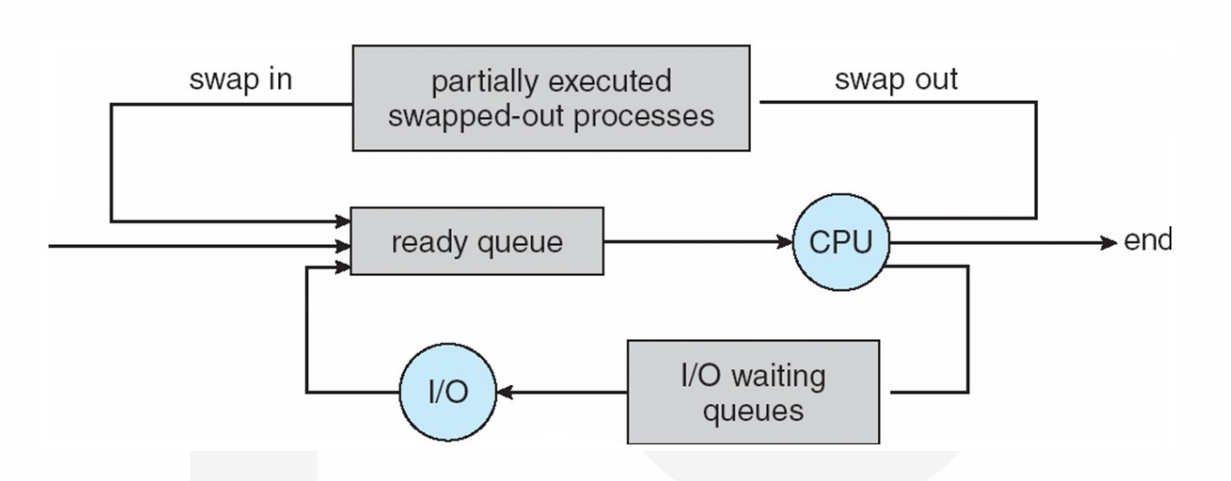
* Runs **less** **frequently**.
* It decides which program must be loaded into the **job queue**.
* Then job processor selects processes from **job queue** and loads them into **memory** for execution.
* Job scheduler is mainly used for **efficiently using multi-programming**.

Short term scheduler:-

* Runs **very frequently**.
* Also known as **CPU scheduler**.
* It is mainly used for **enhancing CPU performance** & **execute processes faster**.

**Medium Term Scheduler**

* **Temporarily** keeps **big processes** present in **ready queue** at some other place when there is too much of **load** on the CPU.
* **Swapping:** The big processes getting **restored** in memory for execution.



**Scheduling Criteria**

* **Keep CPU busy**
* **Throughput:** Number of processes completed per unit time.
* **Turnaround time:** Time taken for a process to complete.
* **Waiting time**, ***\*In ready queue\****
* **Load average:** Number of processes in the ready queue.
* **Response time**, ***\*Response to request\****

**Operation on Process**

* Process creation
* Process termination

**CPU Scheduling**

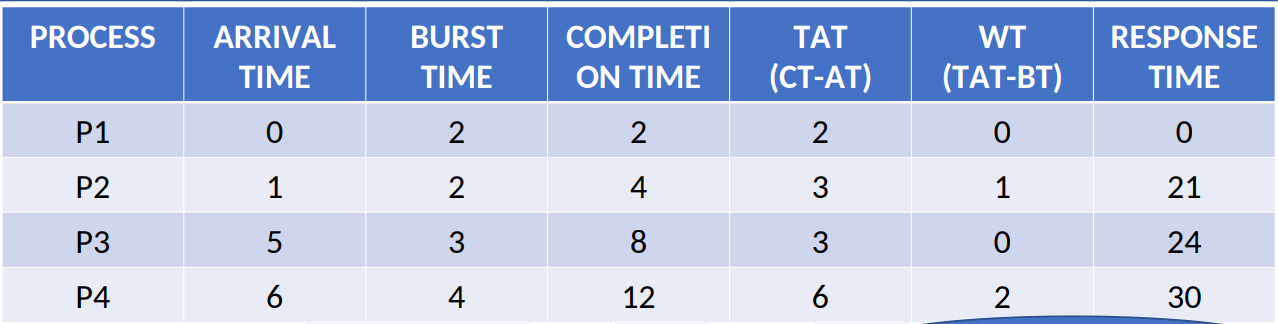
* Processes on hold are there due to **unavailability** **of resources**.
* The **short-term scheduler** selects the **next process** to be carried out.

**Scheduling Algorithms**

* First come first serve scheduling (FCFS)
* Shortest job first scheduling (SJF)
* Priority scheduling
* Round Robin scheduling (RR)
* Multi-level queue scheduling
* Multi-level feedback queue scheduling

**First Come First Serve (FCFS)**

Example:-



* Uses **GANTT chart**.
* ***\*This example has a lot of mistakes.\****

Problems:-

* High waiting time.
* **Convoy effect:** Short process waiting for a long process to complete.
* Poor performance.

**Short Job First (SJF)**

* Of both **pre-emptive** & **non pre-emptive** types.
* Processor need to know the **burst time** of all processes **in advance**.
* Performs better if **arrival time** is **same** for all processes.
* Chooses the **shortest process** among all available as per the arrival time.

Pre-emptive SJF:-

* Jobs are put into **ready queue** as they arrive.
* Job with shorter burst time (if any) **replaces** the current process.

**Fixed Priority Scheduling (FPS)**

Problem With Priority:-

* Small process may wait for a while before a large but **high priority** process ends.
* **Solution:** Keep increasing priority of waiting small processes with time.

**Round Robin Algorithm (RRA)**

* We have to skip time units with **quanta 'q'**.
* Processes get **< 100 milliseconds** of time slice.
* For **large 'q'**, the processes must follow **FIFO**.
* For **small 'q'**, quantum time must be **> context switch time**.
* Context switching takes **< 10 micro seconds** to be done.

**Turn around time = Completion time - Arrival time**

**Waiting time = Turn around time - Burst time**

**Multilevel Queue Scheduling (MQS)**

* Process of **dividing** the **ready queue** into multiple queues.
* Similar processes are assigned to one of those queues permanently.
* **Foreground (front-end)** processes queues follow **RRA**.
* **Background (back-end)** processes queues follow **FCFS**.
* There is even scheduling **among** the queues.
* Some queues are given **higher priority** over others.
* Foreground queues are given **higher priority** than background queues.

**Multilevel Feedback Queue Scheduling (MFQS)**

* Similar to **MQS** but processes can **switch between the queues** as per their behaviour.
* For example, if a process in a queue of **q=8 millisecond** completes itself in **16 milliseconds**, then it is **shifted** to another queue.
* **MFQS parameter:** Methods and variables defined for managing **MFQS**.