UNIT 2 –PROCESS MANAGEMENT

THREAD

✓ **Thread is a sequential flow of tasks within a process.**

✓ Each thread has its own program counter, stack, and set of registers.

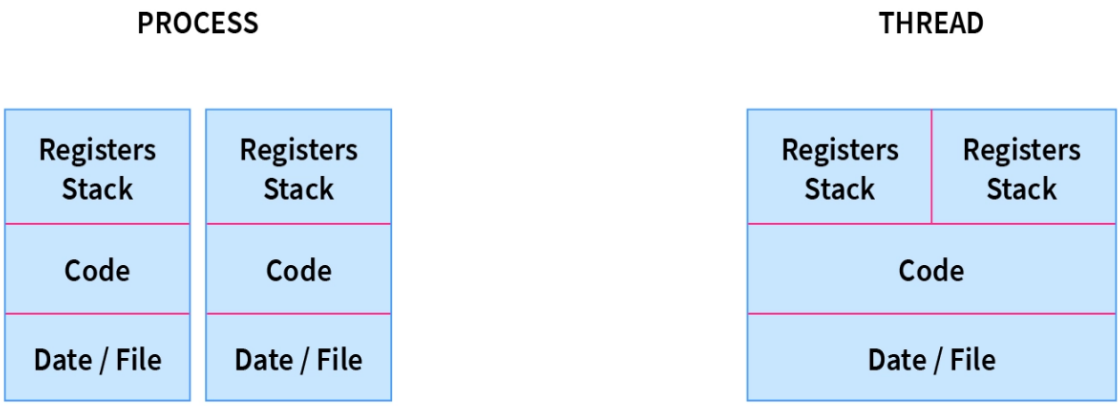
✓ Threads in OS can be of the same or different types.

✓ But the threads of a single process might share the same code and data/file.

✓ Threads are also termed **as lightweight processes as they share common resources**.

✓ Threads provide a way to improve application performance through parallelism.

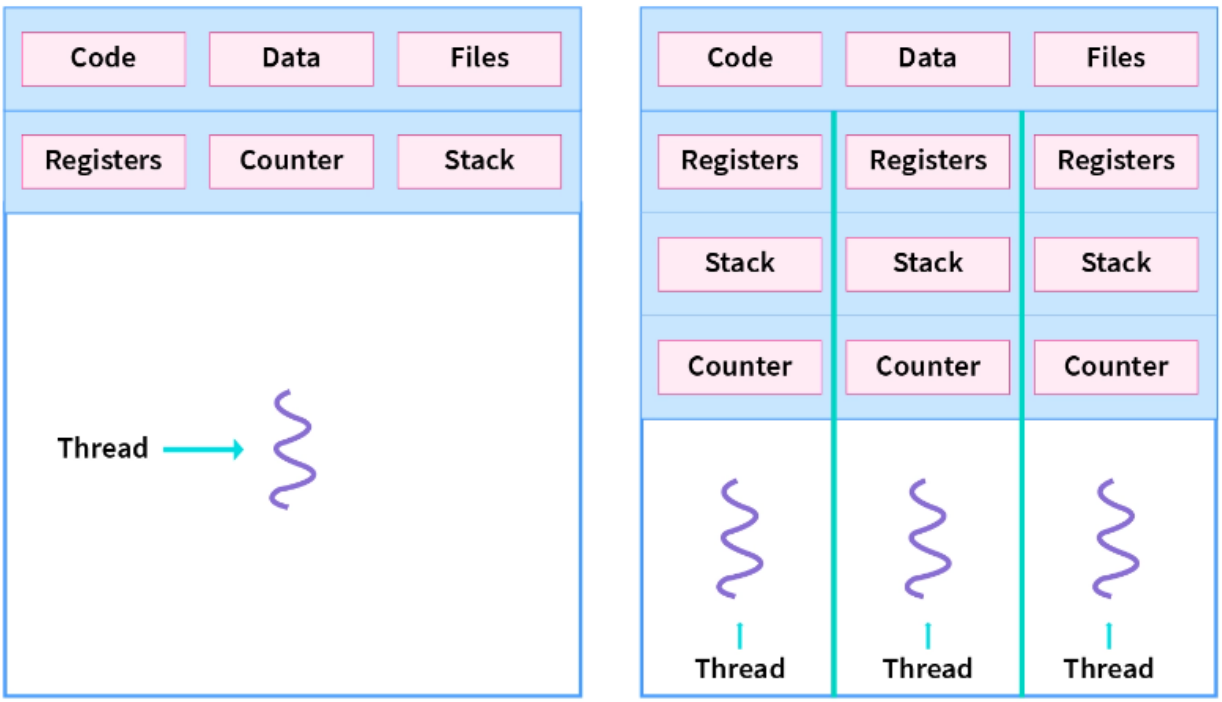
Eg: While playing a movie on a device the audio and video are controlled by different threads in the background.



COMPONENTS OF THREAD

A thread has the following three components:

1. **Program Counter**- keeps track of which instruction to execute next
2. **Register Set**- current working variables
3. **Stack space**- execution history.



Single-threaded process Multithreaded process

Thread Control Blocks (TCBs) represents threads generated in the system. It contains information about the threads, such as it's ID and states.

Thread ID
Thread state
CPU information : Program counter Register contents
Thread priority
Pointer to process that created this thread
Pointer(s) to other thread(s) that were created by this thread

The components have been defined below:

- ✓ **Thread ID:** It is a **unique identifier assigned** by the Operating System to the thread when it is being created.
- ✓ **Thread states:** These are the **states of the thread** which changes as the thread progresses through the system
- ✓ **CPU information:** It includes everything that the OS needs to know about, such as how far the thread has progressed and what data is being used.
- ✓ **Thread Priority:** It indicates the **priority of the thread** over other threads which helps the thread scheduler to determine which thread should be selected next from the READY queue.
- ✓ A **pointer** which **points to the process** which triggered the creation of this thread.
- ✓ A **pointer** which **points to the thread**(s) created by this thread.

BENEFITS OF THREAD

- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.
- Resource sharing.
- *Enhanced throughput of the system*

WHY MULTITHREADING?

- ✓ In Multithreading, single process is divided into **multiple threads** instead of creating a whole new process.
- ✓ Multithreading is done to achieve parallelism and to improve the performance of the applications as it is faster in many ways

ADVANTAGES OF MULTITHREADING

- ✓ **Resource Sharing:** Threads of a single process share the same resources such as code, data/file.
- ✓ **Responsiveness:** Program responsiveness enables a program to run even if part of the program is blocked or executing a lengthy operation. Thus, increasing the responsiveness to the user.

- ✓ **Economy:** It is more economical to use threads as they share the resources of a single process. On the other hand, creating processes is expensive.

DIFFERENCE BETWEEN PROCESS AND THREAD

Process simply means any program in execution while the **thread** is a segment of a process.

Process	**Thread**
Processes use more resources and hence they are termed as heavyweight processes.	Threads share resources and hence they are termed as lightweight processes.
Creation and termination times of processes are slower.	Creation and termination times of threads are faster compared to processes.
Processes have their own code and data/file.	Threads share code and data/file within a process.
Communication between processes is slower.	Communication between threads is faster.
Context Switching in processes is slower.	Context switching in threads is faster.
Processes are independent of each other.	Threads, on the other hand, are interdependent. (i.e they can

Process	Thread
	read, write or change another thread's data)
Eg: Opening two different browsers.	Eg: Opening two tabs in the same browser.

THREAD STATES

When a thread moves through the system, it is always in one of the five states:

(1) Ready

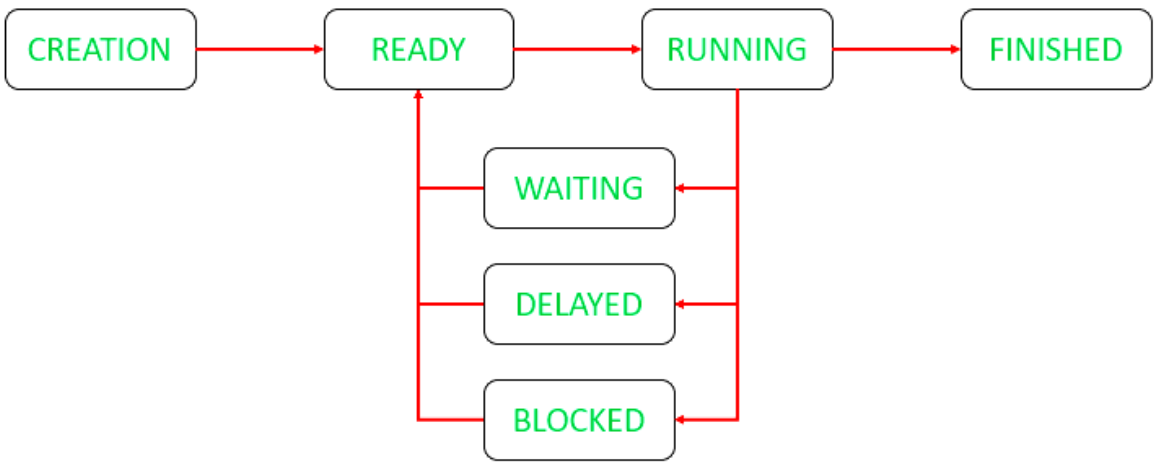
(2) Running

(3) Waiting

(4) Delayed

(5) Blocked

Excluding CREATION and FINISHED state.



1. When an application is to be processed, then it creates a thread.
2. It is then allocated the required resources(such as a network) and it comes in the **READY** queue.
3. When the thread scheduler (like a process scheduler) assign the thread with processor, it comes in **RUNNING** queue.
4. When the process needs some other event to be triggered, which is outsides it's control (like another process to be completed), it transitions from **RUNNING** to **WAITING** queue.
5. When the application has the capability to delay the processing of the thread, it when needed can delay the thread and put it to sleep for a specific amount of time. The thread then transitions from **RUNNING** to **DELAYED** queue.

   An **example of delaying of thread is snoozing of an alarm**. After it rings for the first time and is not switched off by the user, it rings again after a specific amount of time. During that time, the thread is put to sleep.
6. When thread generates an I/O request and cannot move further till it's done, it transitions from **RUNNING** to **BLOCKED** queue.
7. After the process is completed, the thread transitions from **RUNNING** to **FINISHED**.

- ✓ The difference between the **WAITING** and **BLOCKED** transition is that in WAITING the thread waits for the signal from another thread or waits for another process to be completed, meaning the burst time is specific. While, in BLOCKED state, there is no specified time (it depends on the user when to give an input).
- ✓ In order to execute all the processes successfully, the processor needs to maintain the information about each thread through **Thread Control Blocks (TCB)**.

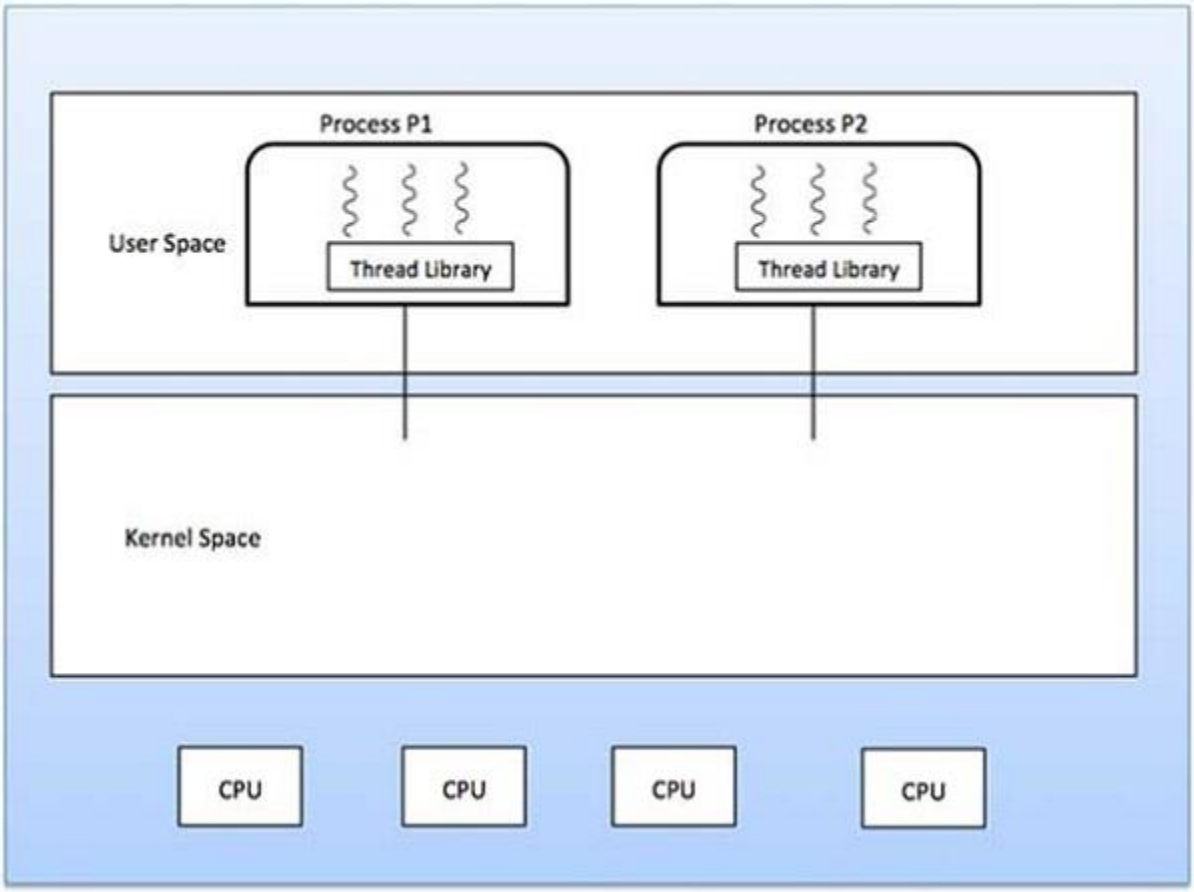
TYPES OF THREAD

Threads are implemented in following two ways −

- **User Level Threads** − User managed threads.
- **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

USER LEVEL THREADS

- ✓ In this case, the thread management kernel is not aware of the existence of threads.
- ✓ The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



Advantages

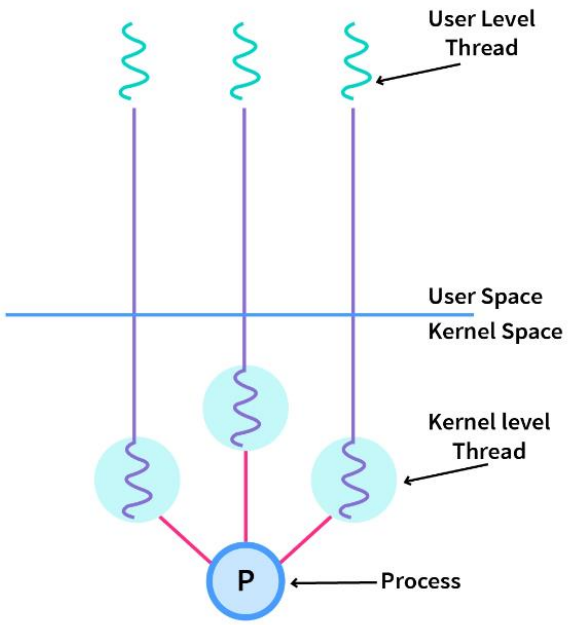
- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

KERNEL LEVEL THREADS

- ✓ In this case, thread management is done by the Kernel. There is no thread management code in the application area.
- ✓ Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded.
- ✓ All of the threads within an application are supported within a single process.
- ✓ The Kernel maintains context information for the process as a whole and for individuals threads within the process.
- ✓ Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space.
- ✓ Kernel threads are generally slower to create and manage than the user threads.



Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

MULTITHREADING MODELS

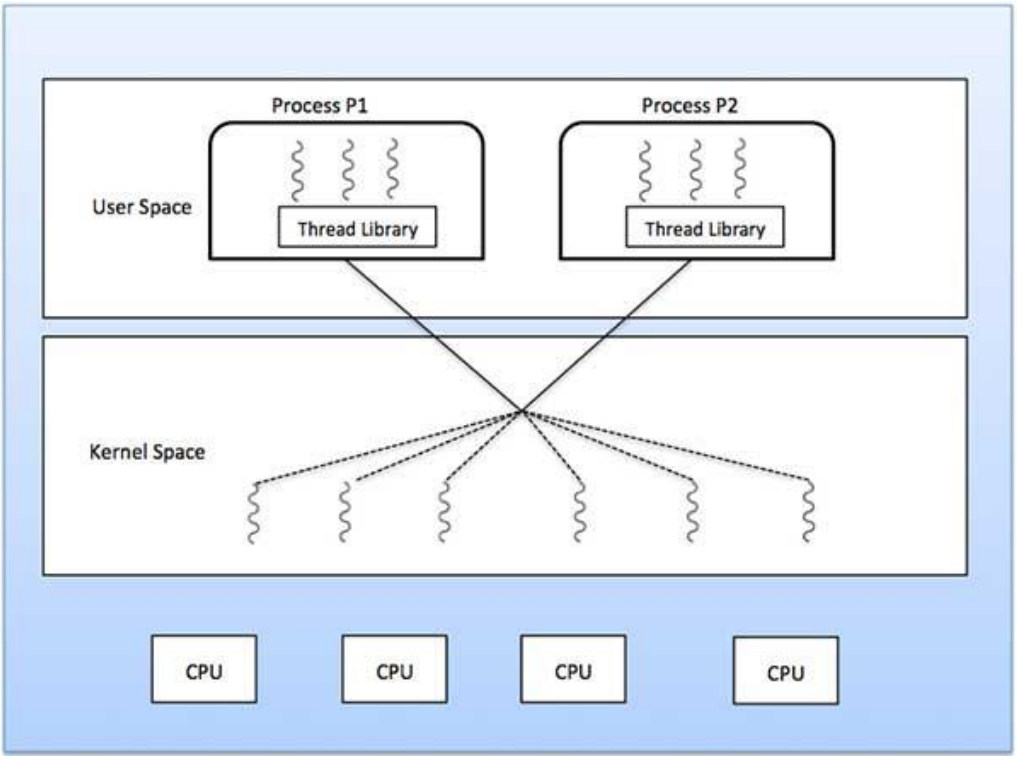
- ✓ Some operating system provide a combined user level thread and Kernel level thread facility.
- ✓ Solaris is a good example of this combined approach.
- ✓ In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.

Multithreading models are three types

- ✓ Many to many relationship.
- ✓ Many to one relationship.
- ✓ One to one relationship.

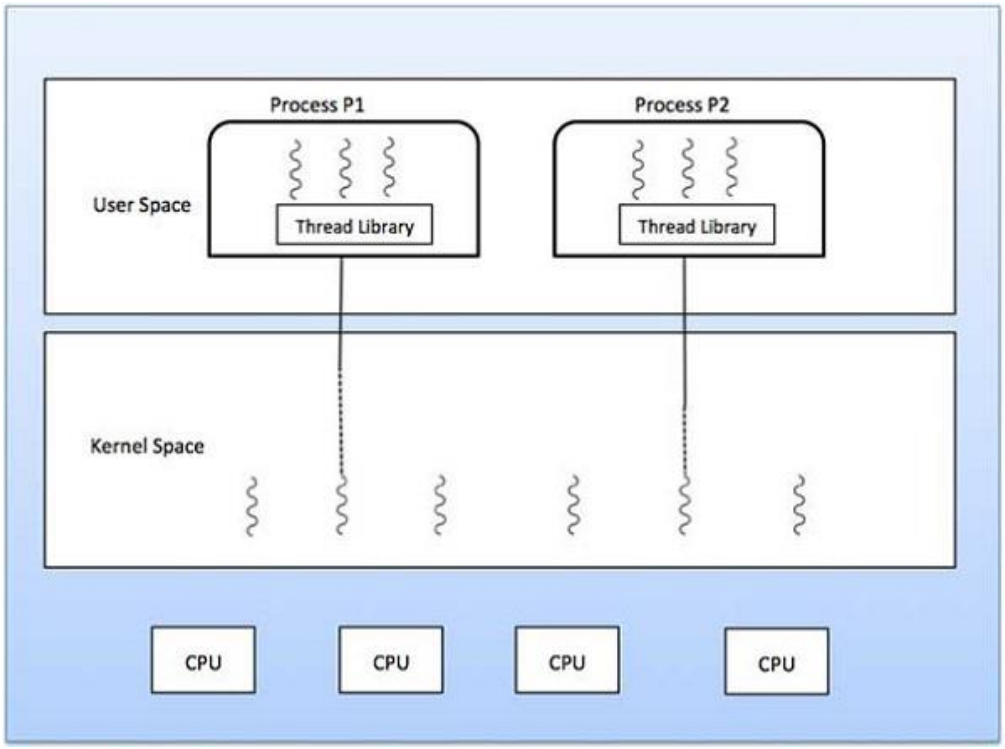
Many to Many Model

- ✓ The many-to-many model multiplexes any **number of user threads onto an equal or smaller number of kernel threads**.
- ✓ In this model, developers can create as **many user threads** as necessary and the corresponding **Kernel threads** can **run in parallel on a multiprocessor machine**.
- ✓ This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.
- ✓ The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads.



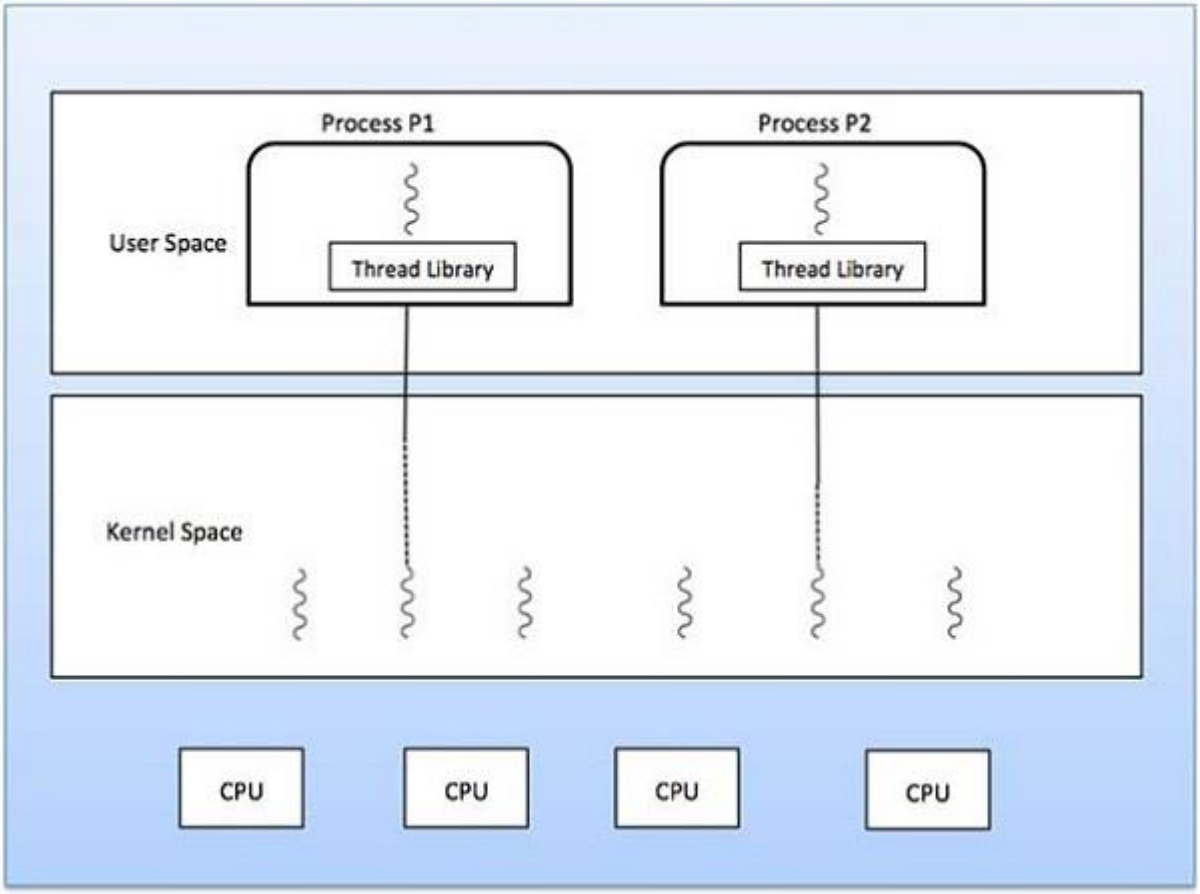
Many to One Model

- ✓ Many-to-one model **maps many user level threads to one Kernel-level thread.**
- ✓ Thread management is done in user space by the thread library.
- ✓ When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.
- ✓ If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



One to One Model

- ✓ There is **one-to-one relationship of user-level thread to the kernel-level thread.**
- ✓ This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call.
- ✓ It supports **multiple threads to execute in parallel on microprocessors.**
- ✓ Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



Difference between User-Level & Kernel-Level Thread

User-Level Threads	Kernel-Level Thread
User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

PROCESS SCHEDULING

- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- Process scheduling is an essential part of a Multiprogramming operating systems.

SCHEDULING OBJECTIVES

- ✓ Be Fair
- ✓ Maximize throughput
- ✓ Maximize number of users receiving acceptible response times.
- ✓ Be predictable
- ✓ Balance resource use
- ✓ Avoid indefinite postponement
- ✓ Enforce Priorities
- ✓ Give preference to processes holding key resources
- ✓ Give better service to processes that have desirable behaviour patterns
- ✓ Degrade gracefully under heavy loads

Different CPU scheduling algorithms have different properties and the choice of a particular algorithm depends on the various factors. Many criteria have been suggested for comparing CPU scheduling algorithms.

SCHEDULING CRITERIA

- **CPU utilisation**

  The main objective of any CPU scheduling algorithm is to keep the CPU as busy as possible. Theoretically, CPU utilisation can range from 0 to 100 but in a real-time system, it varies from 40 to 90 percent depending on the load upon the system.

- **Throughput**

  A measure of the work done by CPU is the number of processes being executed and completed per unit time. This is called throughput. The throughput may vary depending upon the length or duration of processes.

- **Turnaround time**

  For a particular process, an important criteria is how long it takes to execute that process. The time elapsed from the time of submission of a process to the time of completion is known as the turnaround time. Turn-around time is the sum of times spent waiting to get into memory, waiting in ready queue, executing in CPU, and waiting for I/O.

- **Waiting time**

  A scheduling algorithm does not affect the time required to complete the process once it starts execution. It only affects

the waiting time of a process i.e. time spent by a process waiting in the ready queue.

- **Response time**

  In an interactive system, turn-around time is not the best criteria. A process may produce some output fairly early and continue computing new results while previous results are being output to the user. Thus another criteria is the time taken from submission of the process of request until the first response is produced. This measure is called response time.

Arrival Time: Time at which the process arrives in the ready queue.

Completion Time: Time at which process completes its execution.

Burst Time: Time required by a process for CPU execution.

Turn Around Time: Time Difference between completion time and arrival time.

Turn Around Time = Completion Time – Arrival Time

Waiting Time(W.T): Time Difference between turn around time and burst time.

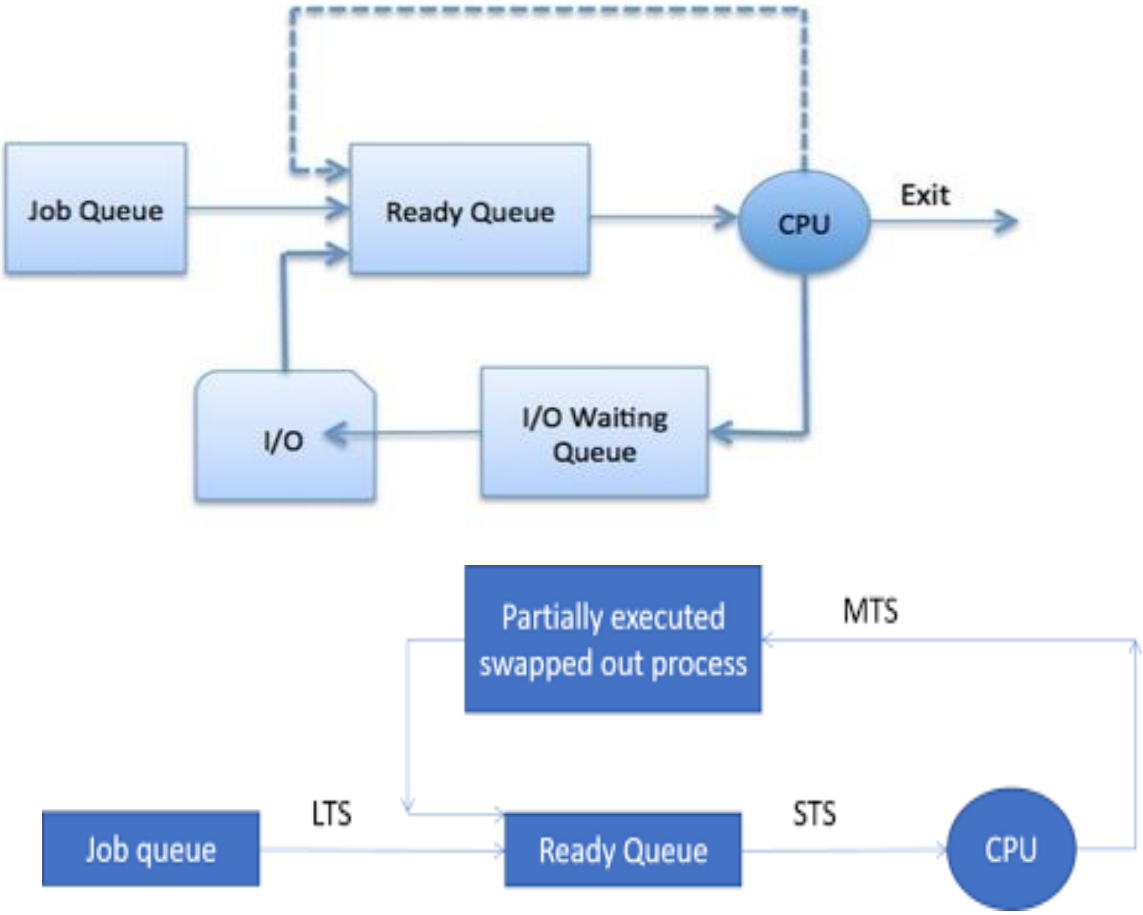
Waiting Time = Turn Around Time – Burst Time

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.
- **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 due to unavailability of an I/O device constitute this queue.



SCHEDULERS

- Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run.
- 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.

Schedulers are of three types –

- ✓ Long-Term Scheduler
- ✓ Short-Term Scheduler
- ✓ Medium-Term Scheduler

LONG TERM SCHEDULER

- ✓ 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. Time-sharing 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.

SHORT TERM SCHEDULER

- ✓ 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.

MEDIUM TERM SCHEDULER

- ✓ 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.

COMPARISON AMONG SCHEDULER

Long-Term Scheduler	**Short-Term Scheduler**	**Medium-Term Scheduler**
It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.

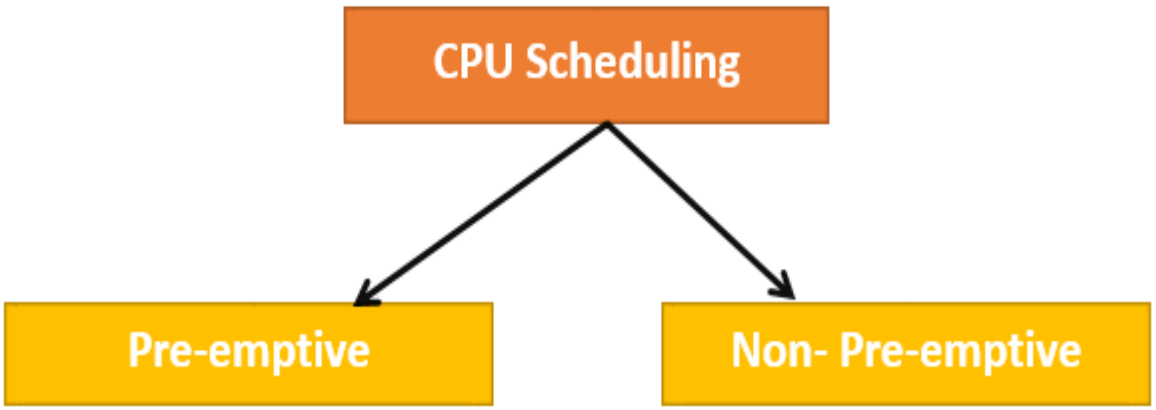
It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

CPU SCHEDULING

CPU Scheduling is a **process of determining which process will own CPU for execution** while another process is on hold. The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution. The selection process will be carried out by the CPU scheduler. It selects one of the processes in memory that are ready for execution.

Types of CPU Scheduling

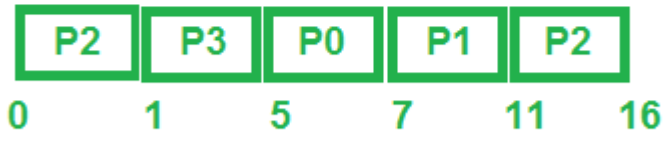
Here are two kinds of Scheduling methods:



Preemptive Scheduling

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

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



Preemptive Scheduling

Non-Preemptive Scheduling

- ✓ In this type of scheduling method, the CPU has been allocated to a specific process.
- ✓ The process that keeps the CPU busy will release the CPU either by switching context or terminating.

- ✓ It is the only method that can be used for various hardware platforms.
- ✓ That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4

0 6 10 14 16

Non-Preemtive Scheduling

When scheduling is Preemptive or Non-Preemptive?

To determine if scheduling is preemptive or non-preemptive, consider these four parameters:

1. A process switches from the running to the waiting state.
2. Specific process switches from the running state to the ready state.
3. Specific process switches from the waiting state to the ready state.
4. Process finished its execution and terminated.

SCHEDULING ALGORITHMS

There are various algorithms which are used by the Operating System to schedule the processes on the processor in an efficient way.

The **Purpose of a Scheduling algorithm**

1. Maximum CPU utilization
2. Fare allocation of CPU
3. Maximum throughput
4. Minimum turnaround time
5. Minimum waiting time
6. Minimum response time

WHY DO WE NEED SCHEDULING?

A typical process involves both I/O time and CPU time. In a uni programming system like MS-DOS, time spent waiting for I/O is wasted and CPU is free during this time. In multi programming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

SCHEDULING ALGORITHMS

- ✓ ***First Come First Serve (FCFS)***
- ✓ ***Shortest Job First (Non-preemptive)***
- ✓ ***Shortest Job First (preemptive)***
- ✓ **Priority Scheduling**
- ✓ **Round robin Scheduling**

FIRST COME FIRST SERVE (FCFS)

- ✓ As the name suggests, the process coming first in the ready state will be executed first by the CPU irrespective of the burst time or the priority.
- ✓ This is implemented by using the **First In First Out (FIFO)** queue.
- ✓ It is a **non-preemptive, pre-emptive scheduling algorithm**.
- ✓ Easy to understand and implement.
- ✓ Its implementation is based on FIFO queue.
- ✓ Poor in performance as average wait time is high.

Example:

Process	Arrival time	Burst time
P1	0 ms	18 ms
P2	2 ms	7 ms
P3	2 ms	10 ms

Gantt Chart

P1		P2		P3	
0 ms	18 ms	18 ms	25 ms	25 ms	35 ms

Explanation

- ✓ In the above example, you can see that we have three processes P1, P2, and P3, and they are coming in the ready state at 0ms, 2ms, and 2ms respectively.
- ✓ So, **based on the arrival time**, the process P1 will be executed for the first 18ms.
- ✓ After that, the process P2 will be executed for 7ms and finally, the process P3 will be executed for 10ms.
- ✓ One thing to be noted here is that if the arrival time of the processes is the same, then the CPU can select any process.

- **Turn Around time = Exit time – Arrival time**
- **Waiting time = Turn Around time – Burst time**

Process Id	Arrival Time	Exit time	Turn Around time	Waiting time
P1	0ms	18	18– 0 = 10	18 – 18 = 0ms
P2	2ms	25	25 – 2 = 23	13 – 7 = 16ms
P3	2ms	35	35 – 2 = 33	33 –10= 23ms

Total waiting time: (0 + 16 + 23) = 39ms

Average waiting time: (39/3) = 13ms

Total turnaround time: (18 + 23 + 33) = 74ms

Average turnaround time: (74/3) = 24.66ms

Advantages of FCFS:

- It is the most simple scheduling algorithm and is easy to implement.

Disadvantages of FCFS:

- This algorithm is non-preemptive so you have to execute the process fully and after that other processes will be allowed to execute.
- Throughput is not efficient.
- FCFS suffers from the **Convey effect** i.e. if a process is having very high burst time and it is coming first, then it will be executed

first irrespective of the fact that a process having very less time is there in the ready state.

SHORTEST JOB FIRST (NON-PREEMPTIVE)

- ✓ **In the FCFS**, we saw if a **process is having a very high burst time and it comes first** then the **other process with a very low burst time have to wait for its turn**. So, to remove this problem, we come with a new approach i.e. Shortest Job First or SJF.
- ✓ In this technique, the process having the minimum burst time at a particular instant of time will be executed first. if the process starts its execution then it will be fully executed and then some other process will come.This is also known as **shortest job first**, or SJF
- ✓ This is a **non-preemptive, pre-emptive** scheduling algorithm.
- ✓ Best approach to minimize waiting time.
- ✓ Easy to implement in Batch systems where required CPU time is known in advance.
- ✓ Impossible to implement in interactive systems where required CPU time is not known.
- ✓ The processer should know in advance how much time process will take.

Example

Process	Arrival time	Burst time
P1	3 ms	5 ms
P2	0 ms	4 ms
P3	4 ms	2 ms
P4	5 ms	4 ms

Gantt Chart

P2	P3	P4	P1
0ms 4ms	4ms 6ms	6ms 10ms	10ms 15ms

Explanation:

- ✓ In the above example, at 0ms, we have only one process i.e. process P2, so the process P2 will be executed for 4ms.
- ✓ Now, after 4ms, there are two new processes i.e. process P1 and process P3. The burst time of P1 is 5ms and that of P3 is 2ms.
- ✓ So, amongst these two, the process P3 will be executed first because its burst time is less than P1.
- ✓ P3 will be executed for 2ms. Now, after 6ms, we have two processes with us i.e. P1 and P4 (because we are at 6ms and P4 comes at 5ms).

✓ Amongst these two, the process P4 is having a less burst time as compared to P1. So, P4 will be executed for 4ms and after that P1 will be executed for 5ms.

- **Turn Around time = Exit time – Arrival time**
- **Waiting time = Turn Around time – Burst time**

Process Id	Arrival Time	Exit time	Turn Around time	Waiting time
P1	3ms	15	15– 3 = 12ms	12 – 5 = 7ms
P2	0ms	4	4 – 0 = 4ms	4 – 4= 0ms
P3	4ms	6	6 – 4= 2ms	2 –2= 0ms
P4	5ms	10	10-5=ms	5-4=1ms

Total waiting time: (7 + 0 + 0 + 1) = 8ms

Average waiting time: (8/4) = 2ms

Total turnaround time: (12 + 4 + 2 + 5) = 23ms

Average turnaround time: (23/4) = 5.75ms

Advantages of SJF (non-preemptive):

- Short processes will be executed first.

Disadvantages of SJF (non-preemptive):

- It may lead to starvation if only short burst time processes are coming in the ready state(learn more about starvation from here).

Shortest Job First (Preemptive)

- ✓ This is the **preemptive approach** of the Shortest Job First algorithm. Here, at every instant of time, the CPU will check for some shortest job.
- ✓ For example, at time 0ms, we have P1 as the shortest process. So, P1 will execute for 1ms and **then the CPU will check if some other process is shorter than P1 or not.**
- ✓ If there is no such process, then P1 will keep on executing for the next 1ms and **if there is some process shorter than P1 then that process will be executed**. This will continue until the process gets executed.
- ✓ *This algorithm is also known as Shortest Remaining Time First i.e. we schedule the process based on the shortest remaining time of the processes.*

Example:

Process	Arrival time	Burst time
P1	1 ms	6 ms
P2	1 ms	8 ms
P3	2 ms	7 ms
P4	3 ms	3 ms

Gantt Chart

P1		P4		P1		P3		P2	
1	3	3	6	6	10	10	17	17	25

EXPLANATION

- ✓ In the above example, at time 1ms, there are two processes i.e. P1 and P2. Process P1 is having burst time as 6ms and the process P2 is having 8ms. So, P1 will be executed first. Since it is a preemptive approach, so we have to check at every time quantum.
- ✓ At 2ms, we have three processes i.e. P1(5ms remaining), P2(8ms), and P3(7ms). Out of these three, P1 is having the least burst time, so it will continue its execution.
- ✓ After 3ms, we have four processes i.e P1(4ms remaining), P2(8ms), P3(7ms), and P4(3ms). Out of these four, P4 is having the least burst time, so it will be executed.
- ✓ The process P4 keeps on executing for the next three ms because it is having the shortest burst time.
- ✓ After 6ms, we have 3 processes i.e. P1(4ms remaining), P2(8ms), and P3(7ms). So, P1 will be selected and executed. This process of time comparison will continue until we have all the processes executed.

- **Turn Around time = Exit time – Arrival time**
- **Waiting time = Turn Around time – Burst time**

Process Id	Arrival Time	Exit time	Turn Around time	Waiting time
P1	1ms	10	10– 1 = 9ms	9 – 6 = 3ms
P2	1ms	25	25 – 1 = 24ms	24 – 8= 16ms
P3	2ms	17	17 – 2= 15ms	15 –7= 8ms
P4	3ms	6	6-3=3ms	3-3=0ms

Total waiting time: (3 + 16 + 8 + 0) = 27ms

Average waiting time: (27/4) = 6.75ms

Total turnaround time: (9 + 24 + 15 + 3) = 51ms

Average turnaround time: (51/4) = 12.75ms

Advantages of SJF (preemptive):

- Short processes will be executed first.

Disadvantages of SJF (preemptive):

- It may result in starvation if short processes keep on coming.

PRIORITY SCHEDULING (NON-PREEMPTIVE)

- ✓ In this approach, we have a priority number associated with each process and based on that priority number the CPU selects one process from a list of processes.
- ✓ Priority scheduling is a **non-preemptive algorithm** and one of the most common scheduling algorithms in batch systems.
- ✓ The priority number can be anything. It is just used to identify which **process is having a higher priority and which process is having a lower priority**.
- ✓ For example, you can denote 0 as the highest priority process and 100 as the lowest priority process. Also, the reverse can be true i.e. you can denote 100 as the highest priority and 0 as the lowest priority.
- ✓ Processes with same priority are executed on first come first served basis.
- ✓ **Priority can be decided based on memory requirements, time requirements or any other resource requirement.**

Example:

Process	Arrival time	Burst time	Priority
P1	0 ms	5 ms	1
P2	1 ms	3 ms	2
P3	2 ms	8 ms	1
P4	3 ms	6 ms	3

NOTE: In this example, we are taking higer priority number as higher priority.

Gantt Chart

P1	P4	P2	P3
0ms 5ms	5ms 11ms	11ms 14ms	14ms 22ms

EXPLANATION

- ✓ In the above example, at 0ms, we have only one process P1. So P1 will execute for 5ms because we are using **non-preemption technique here.**
- ✓ After 5ms, there are three processes in the ready state i.e. process P2, process P3, and process P4. Out to these three processes, the process P4 is having the highest priority so it will be executed for 6ms .
- ✓ After that, process P2 will be executed for 3ms followed by the process P1.

- **Turn Around time = Exit time – Arrival time**
- **Waiting time = Turn Around time – Burst time**

Process Id	Arrival Time	Exit time	Turn Around time	Waiting time
P1	0ms	5ms	5-0 = 5ms	5 – 5 = 0ms
P2	1ms	14ms	14-1 = 13ms	13 – 3= 10ms
P3	2ms	22ms	22 – 2= 20ms	20 –8= 12ms
P4	3ms	11ms	11-3=8ms	8-6=2ms

Total waiting time: (0 + 10 + 12 + 2) = 24ms

Average waiting time: (24/4) = 6ms

Total turnaround time: (5 + 13 + 20 + 8) = 46ms

Average turnaround time: (46/4) = 11.5ms

Advantages of priority scheduling (non-preemptive):

- Higher priority processes like system processes are executed first.

Disadvantages of priority scheduling (non-preemptive):

- It can lead to starvation if only higher priority process comes into the ready state.
- If the priorities of more two processes are the same, then we have to use some other scheduling algorithm.

ROUND-ROBIN

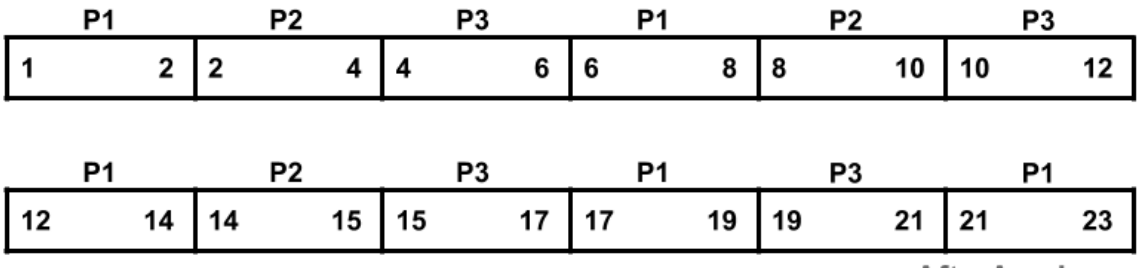
- ✓ Round Robin is the **preemptive process scheduling algorithm**.
- ✓ Each process is provided a fix time to execute, it is called a **quantum**.
- ✓ Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- ✓ Context switching is used to save states of preempted processes.
- ✓ It is generally used in the time-sharing environments and there will be no starvation in case of the round-robin.

For example, if we are having three process P1, P2, and P3, and our time quantum is 2ms, then P1 will be given 2ms for its execution, then P2 will be given 2ms, then P3 will be given 2ms. After one cycle, again P1 will be given 2ms, then P2 will be given 2ms and so on until the processes complete its execution.

Example:

Process	Arrival time	Burst time
P1	0 ms	10 ms
P2	0 ms	5 ms
P3	0 ms	8 ms

Gantt Chart



EXPLANATION

- ✓ In the above example, every process will be given 2ms in one turn because we have taken the time quantum to be 2ms. So process P1 will be executed for 2ms, then process P2 will be executed for 2ms, then P3 will be executed for 2 ms.
- ✓ Again process P1 will be executed for 2ms, then P2, and so on.

- **Turn Around time = Exit time – Arrival time**
- **Waiting time = Turn Around time – Burst time**

Process Id	Arrival Time	Exit time	Turn Around time	Waiting time
P1	0ms	23ms	23-0 = 23ms	23 – 10 = 13ms
P2	0ms	15ms	15-0 = 15ms	15 – 5= 10ms
P3	0ms	21ms	21 – 0= 21ms	21 –8= 13ms

Total waiting time: (13 + 10 + 13) = 36ms

Average waiting time: (36/3) = 12ms

Total turnaround time: (23 + 15 + 21) = 59ms

Average turnaround time: (59/3) = 19.66ms

Advantages of round-robin:

- No starvation will be there in round-robin because every process will get chance for its execution.
- Used in time-sharing systems.

Disadvantages of round-robin:

- We have to perform a lot of context switching here, which will keep the CPU idle(learn more about context switching from here).