



**CHANDIGARH**  
**ENGINEERING COLLEGE**  
Building Careers. Transforming Lives.

# Process Scheduling

Dr. Simarpreet Kaur



# Process

A process is basically a program in execution. A process is defined as an entity which represents the basic unit of work to be implemented in the system. In order to accomplish its task, process needs the computer resources. There may exist more than one process in the system which may require the same resource at the same time. Therefore, the operating system has to manage all the processes and the resources in a convenient and efficient way.

When a program is loaded into the memory and it becomes a process, it can be divided into  $0$  heap, text and data. **Process memory** is divided into four sections for efficient working.

1	<b>Stack</b> The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2	<b>Heap</b> This is dynamically allocated memory to a process during its run time.
3	<b>Text</b> This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4	<b>Data</b> This section contains the global and static variables.

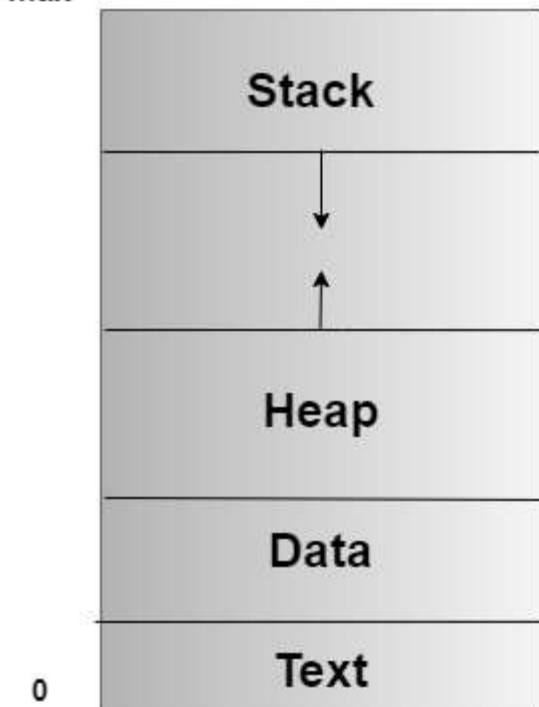
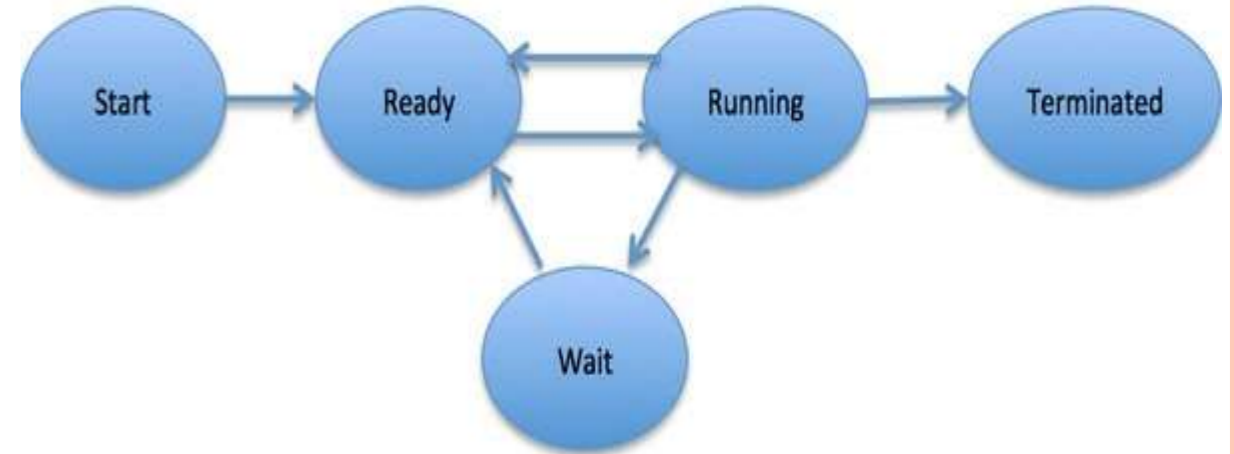


Figure: Process in the Memory

## Process States

1	<b>Start</b> This is the initial state when a process is first started/created.
2	<b>Ready</b> The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after <b>Start</b> state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	<b>Running</b> Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	<b>Waiting</b> Process moves in to the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5	<b>Terminated or Exit</b> Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.



## Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table. The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

### Process State

The current state of the process i.e., whether it is ready, running, waiting, or whatever.

### Process privileges

This is required to allow/disallow access to system resources.

### Process ID

Unique identification for each of the process in the operating system.

### Pointer

A pointer to parent process.

### Program Counter

Program Counter is a pointer to the address of the next instruction to be executed for this process.

### CPU registers

Various CPU registers where process need to be stored for execution for running state.

### CPU Scheduling Information

Process priority and other scheduling information which is required to schedule the process.

### Memory management information

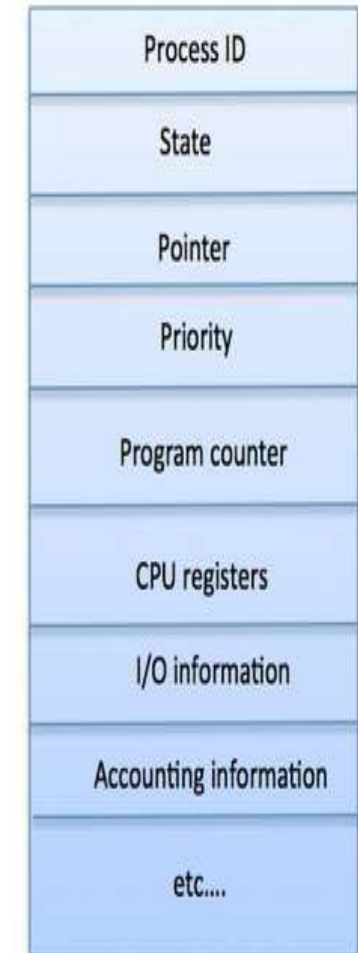
This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.

### Accounting information

This includes the amount of CPU used for process execution, time limits, execution ID etc.

### IO status information

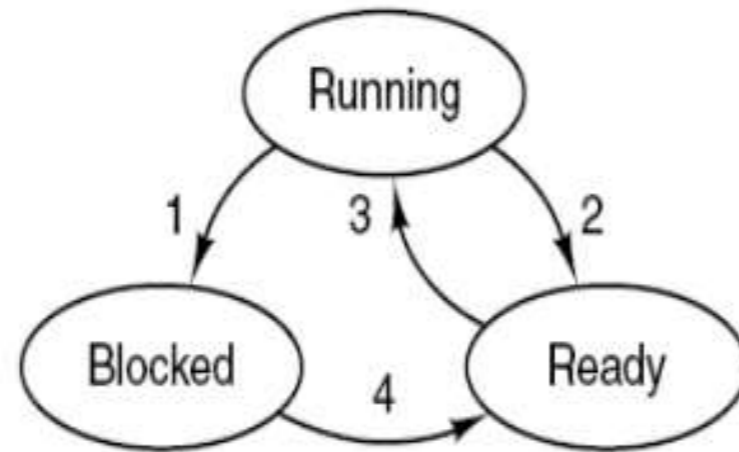
This includes a list of I/O devices allocated to the process.



## State Transition

The process can be in any one of the following three possible states.

- 1) Running (actually using the CPU at that time and running).
- 2) Ready (runnable; temporarily stopped to allow another process run).
- 3) Blocked (unable to run until some external event happens).



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available



## Context switching

The Context switching is a technique or method used by the operating system to switch a process from one state to another to execute its function using CPUs in the system. When switching perform in the system, it stores the old running process's status in the form of registers and assigns the CPU to a new process to execute its tasks. While a new process is running in the system, the previous process must wait in a ready queue. The execution of the old process starts at that point where another process stopped it. It defines the characteristics of a multitasking operating system in which multiple processes shared the same CPU to perform multiple tasks without the need for additional processors in the system.

### Need for Context switching

- A context switching helps the operating system that switches between the multiple processes to use the CPU's resource to accomplish its tasks and store its context. We can resume the service of the process at the same point later. If we do not store the currently running process's data or context, the stored data may be lost while switching between processes.
- If a high priority process falls into the ready queue, the currently running process will be shut down or stopped by a high priority process to complete its tasks in the system.
- If any running process requires I/O resources in the system, the current process will be switched by another process to use the CPUs. And when the I/O requirement is met, the old process goes into a ready state to wait for its execution in the CPU. Context switching stores the state of the process to resume its tasks in an operating system. Otherwise, the process needs to restart its execution from the initials level.
- If any interrupts occur while running a process in the operating system, the process status is saved as registers using context switching. After resolving the interrupts, the process switches from a wait state to a ready state to resume its execution at the same point later, where the operating system interrupted occurs.
- A context switching allows a single CPU to handle multiple process requests simultaneously without the need for any additional processors.

## Context switching triggers

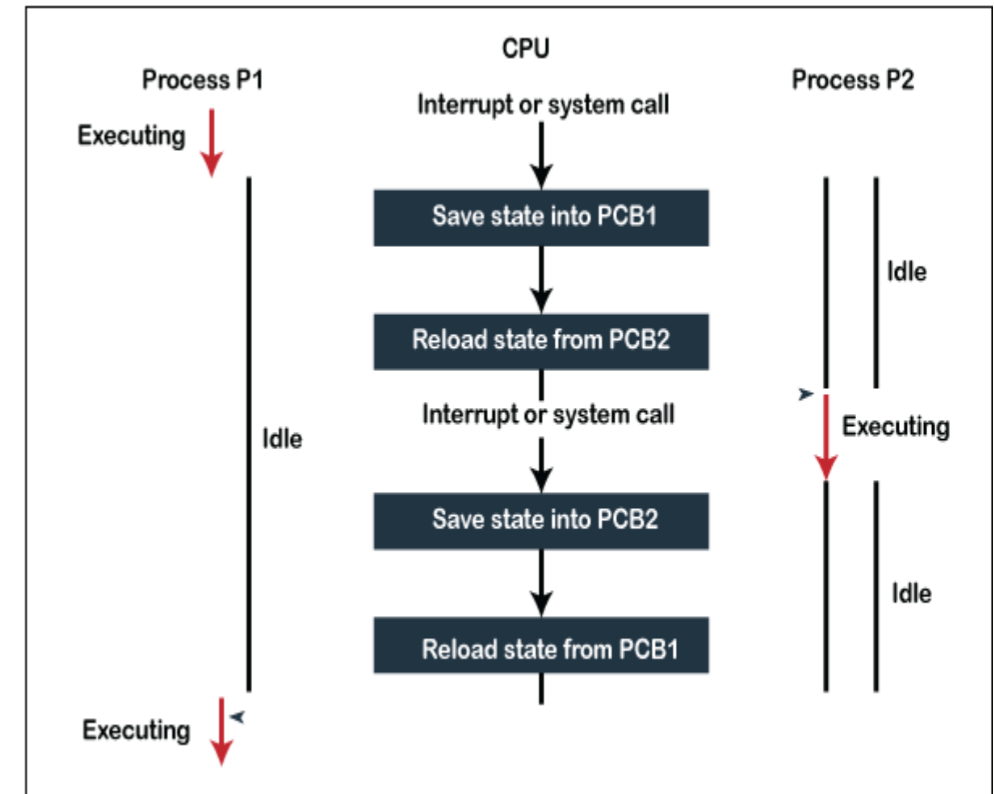
**Interrupts:** A CPU requests for the data to read from a disk, and if there are any interrupts, the context switching automatic switches a part of the hardware that requires less time to handle the interrupts.

**Multitasking:** A context switching is the characteristic of multitasking that allows the process to be switched from the CPU so that another process can be run. When switching the process, the old state is saved to resume the process's execution at the same point in the system.

**Kernel/User Switch:** It is used in the operating systems when switching between the user mode, and the kernel/user mode is performed.

### Steps

- First, the context switching needs to save the state of process P1 in the form of the program counter and the registers to the PCB (Program Counter Block), which is in the running state.
- Now update PCB1 to process P1 and moves the process to the appropriate queue, such as the ready queue, I/O queue and waiting queue.
- After that, another process gets into the running state, or we can select a new process from the ready state, which is to be executed, or the process has a high priority to execute its task.
- Now, we have to update the PCB (Process Control Block) for the selected process P2. It includes switching the process state from ready to running state or from another state like blocked, exit, or suspend.
- If the CPU already executes process P2, we need to get the status of process P2 to resume its execution at the same time point where the system interrupt occurs.



## What is a Thread?

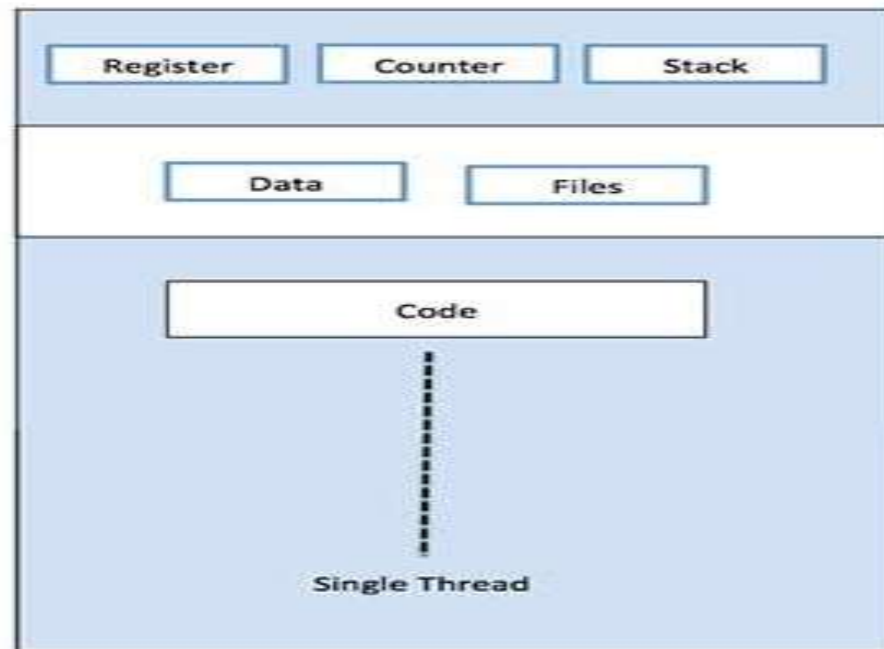
A thread is a path of execution within a process. A process can contain multiple threads. A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads.

For example, in a browser, multiple tabs can be different threads.

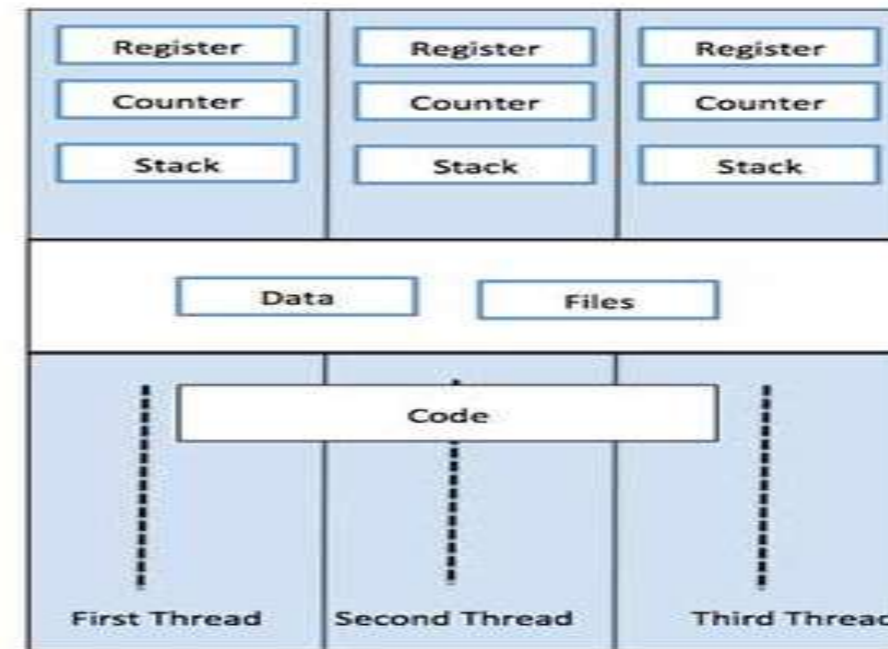
## Process vs Thread?

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.

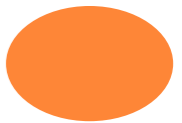
Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.



Single Process P with single thread



Single Process P with three threads



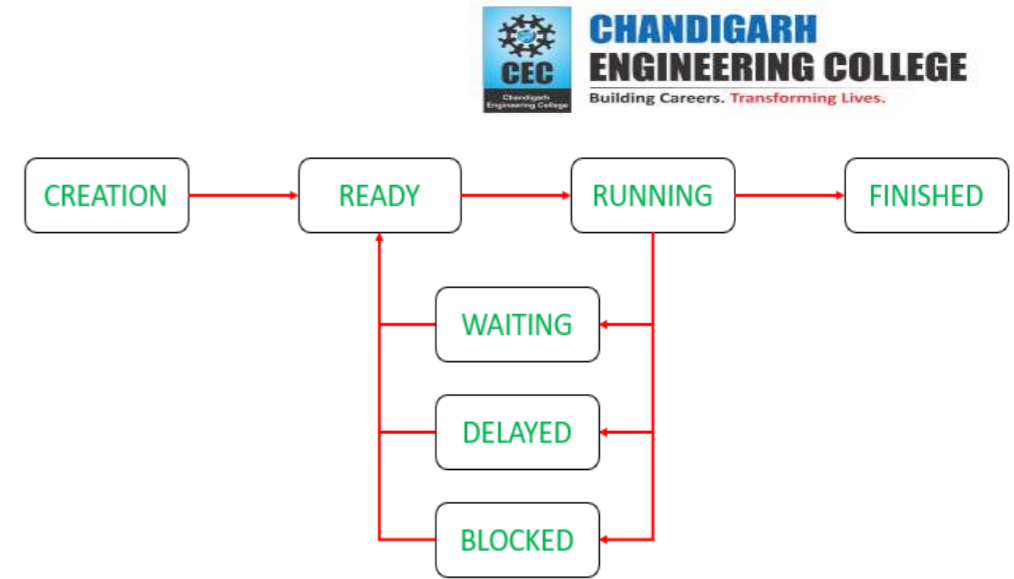
## ***Advantages of Thread over Process***

1. *Responsiveness*: If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
2. *Faster context switch*: Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.
3. *Effective utilization of multiprocessor system*: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.
4. *Resource sharing*: Resources like code, data, and files can be shared among all threads within a process.
5. *Communication*: Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two process.
6. *Enhanced throughput of the system*: If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.



# Thread States in Operating Systems

- When an application is to be processed, then it creates a thread.
- It is then allocated the required resources(such as a network) and it comes in the **READY** queue.
- When the thread scheduler (like a process scheduler) assign the thread with processor, it comes in **RUNNING** queue.
- When the process needs some other event to be triggered, which is outside its control (like another process to be completed), it transitions from **RUNNING** to **WAITING** queue.
- 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.
- When thread generates an I/O request and cannot move further till it's done, it transitions from **RUNNING** to **BLOCKED** queue.
- 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 Threads:**

### **User Level thread (ULT) –**

Is implemented in the user level library, they are not created using the system calls. Thread switching does not need to call OS and to cause interrupt to Kernel. Kernel doesn't know about the user level thread and manages them as if they were single-threaded processes.

- Can be implemented on an OS that doesn't support multithreading.
- Simple representation since thread has only program counter, register set, stack space.
- Simple to create since no intervention of kernel.
- Thread switching is fast since no OS calls need to be made.
- No or less co-ordination among the threads and Kernel.
- If one thread causes a page fault, the entire process blocks.

### **Kernel Level Thread (KLT) –**

Kernel knows and manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. In addition kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system call to create and manage threads. Since kernel has full knowledge about the threads in the system, scheduler may decide to give more time to processes having large number of threads.

- Good for applications that frequently block.
- Slow and inefficient.
- It requires thread control block so it is an overhead.

### **Summary:**

Each ULT has a process that keeps track of the thread using the Thread table.

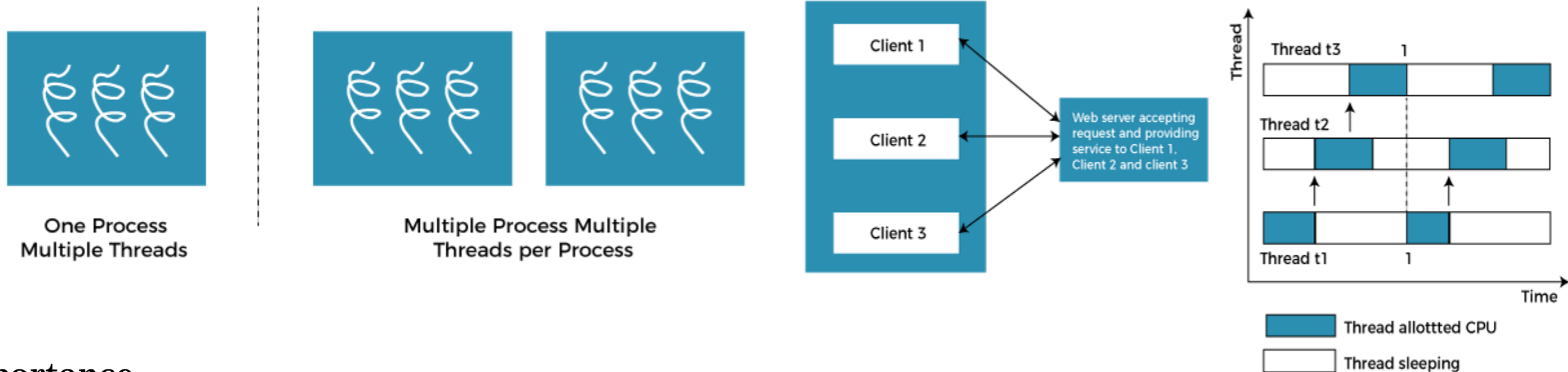
Each KLT has Thread Table (TCB) as well as the Process Table (PCB).



## Multithreading

Multithreading allows the application to divide its task into individual threads. In multi-threads, the same process or task can be done by the number of threads, or we can say that there is more than one thread to perform the task in multithreading. With the use of multithreading, multitasking can be achieved.

The main drawback of single threading systems is that only one task can be performed at a time, so to overcome the drawback of this single threading, there is multithreading that allows multiple tasks to be performed.



## Importance

- Improving Front end response to the users
- Enhancing application performance
- Effective utilization of Computer resources
- Low maintenance cost
- Faster completion of tasks due to parallel operation
- Simplifying development process and increasing productivity

There exists three established multithreading models classifying these relationship

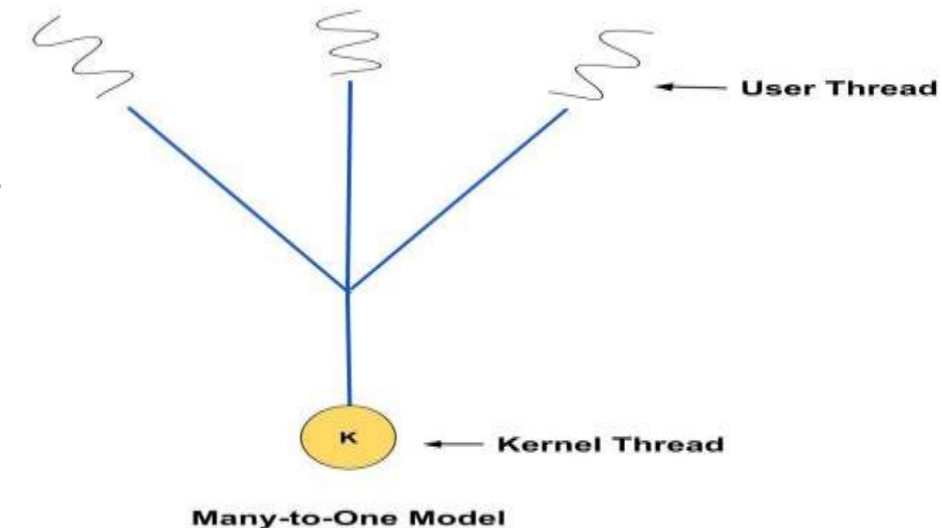
- Many to one multithreading model
- One to one multithreading model
- Many to Many multithreading models

### Many-to-One Model

As the name suggests there is many to one relationship between threads. Here, multiple user threads are associated or mapped with one kernel thread. The thread management is done on the user level so it is more efficient.

### Drawbacks

- As multiple users threads are mapped to one kernel thread. So, if one **user thread** makes a blocking system call( like function read() call then the thread or process has to wait until read event is completed), it will block the **kernel thread** which will in turn block all the other threads.
- As only one thread can access the kernel thread at a time so multiple threads are unable to run in parallel in the multiprocessor system. Even though we have multiple processors **one kernel thread will run on only one processor**. Hence, the user thread will also run in that processor only in which the mapped kernel thread is running.



## One-to-One Model

### •Advantages over Many-to-One Model

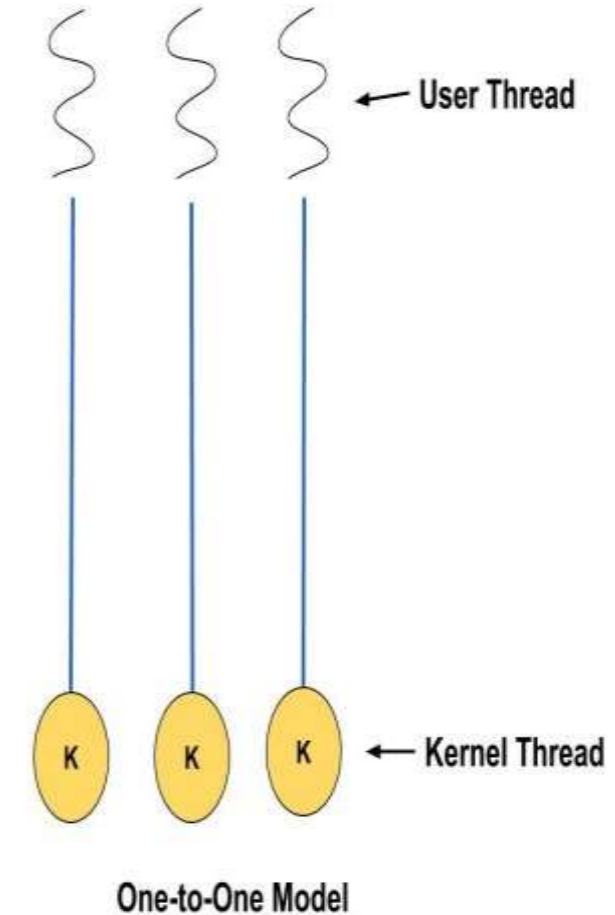
- In this model, the first drawback of the Many-to-One model is solved. As each user thread is mapped to different kernel threads so even if any user thread makes a blocking system call, the other user threads won't be blocked.

- The second drawback is also overcome. It allows the threads to run parallel on a multiprocessor. Since it has each kernel threads mapped to one user thread. So each **kernel thread can run on different processors**. Hence, each user thread can run on one of the processors.

### Disadvantages

- Each time we create a user thread we have to create a kernel thread. So, the overhead of creating a kernel thread can affect the performance of the application.

- Also, in a multiprocessor system, there is a limit of how many threads can run at a time. Suppose if there are four-processors in the system then only max four threads can run at the time. So, if you are having 5 threads and trying to run them at a time then it may not work. Therefore, the application should restrict the number of kernel threads that it supports.



## Many-to-Many Model

### •Advantages over the other two models

- Whenever a user thread makes a blocking system call other threads are not blocked.
- There can be as many user threads as necessary. Also, the threads can run parallel on multiple processors.
- Here we don't need to create as many kernel threads as the user thread. So, there is no problem with any extra overhead which was caused due to creating kernel thread.
- The number of kernel threads supported here is specific to the application or machine.
- So, this is the best model that we can have in a multithreading system to establish the relationship between user-thread and kernel thread.

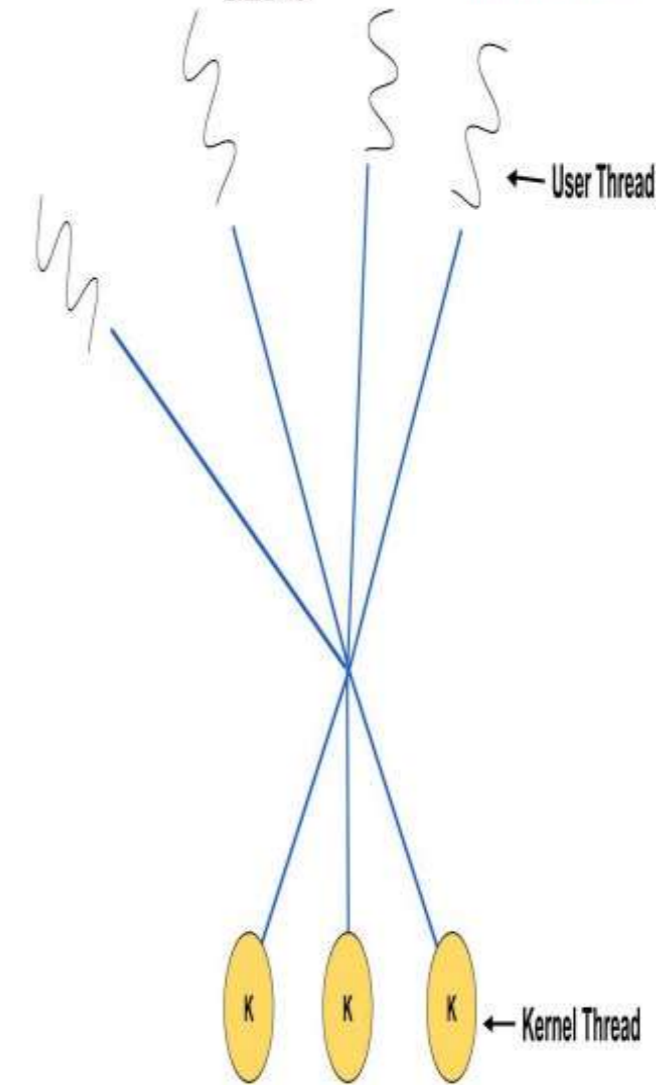
### Benefits of MultiThreading

**Resource sharing:** As the threads can share the memory and resources of any process it allows any application to perform multiple activities inside the same address space.

**Utilization of MultipleProcessor Architecture:** The different threads can run parallel on the multiple processors hence, this enables the utilization of the processor to a large extent and efficiency.

**Reduced Context Switching Time:** The threads minimize the context switching time as in Thread Context Switching, the virtual memory space remains the same.

**Economical:** The allocation of memory and resources during process creation comes with a cost. As the threads can distribute resources of the process it is more economical to create context-switch threads.



Many-to-Many Model

**CPU scheduling** is the foundation or starting concept of multi-programmed operating systems (OSs). By toggling the CPU with different processes, the operating system can make the computer and its processing power more productive.

The CPU scheduling is the action done by the process manager to handle the elimination of the running process within the CPU and the inclusion of another process by certain specific strategies.

In a single-processor system, only one job can be processed at a time; rest of the job must wait until the CPU gets free and can be rescheduled. The aim of multiprogramming is to have some process to run at all times, for maximizing CPU utilization.

### Objectives of Process Scheduling in OS

Following are the objectives of process scheduling:

1. It maximizes the number of interactive users within acceptable response times.
2. It achieves a balance between response and utilization.
3. It makes sure that there is no postponement for an unknown time and enforces priorities.
4. It gives reference to the processes holding the key resources.



## Schedulers in OS

A scheduler is a special type of system software that handles process scheduling in numerous ways. It mainly selects the jobs that are to be submitted into the system and decides whether the currently running process should keep running or not. If not then which process should be the next one to run. A scheduler makes a decision:

- When the state of the current process changes from running to waiting due to an I/O request or some unsatisfied OS.
  - If the current process terminates.
  - When the scheduler needs to move a process from running to ready state as it has already run for its allotted interval of time.
  - When the requested I/O operation is completed, a process moves from the waiting state to the ready state.
- So, the scheduler can decide to replace the currently-running process with a newly-ready one.

### **Long Term Scheduler:**

A long-term scheduler also known as a job scheduler determines which program should be admitted to the system for processing. It selects and loads the processes into the memory for execution with the help of CPU scheduling. Many systems like time-sharing OS, do not have a long term scheduler as it is only required when a process changes its state from new to ready.



### **Short Term Scheduler:**

A short-term scheduler also known as a CPU scheduler increases system performance as per the chosen set of criteria. This is the change of ready state to running state of the process.

It selects a process from the multiple processes that are in ready state in order to execute it and also allocates the CPU to one of them. It is faster than long-term schedulers and is also called a dispatcher as it makes the decision on which process will be executed next.

### **Medium Term Scheduler:**

Medium-term scheduling removes processes from the memory and is a part of swapping.

When a running process makes an I/O request it becomes suspended i.e., it cannot be completed. Thus, in order to remove the process from the memory and make space for others, the suspended process is sent to the secondary storage. This is known as swapping, and the process that goes through swapping is said to be swapped out or rolled out.

### **Role of the dispatcher in CPU scheduling**

The dispatcher gives control of the CPU to the process selected by the short-term scheduler.

In order to perform this task, a context switch, a switch to user mode, and a jump to the proper location in the user program are all required.

The dispatch should be made as fast as possible since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another is known as the *dispatch latency*



## **Scheduling Criteria**

### **CPU Utilization**

To make out the best use of the CPU and not to waste any CPU cycle, the CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

### **Throughput**

It is the total number of processes completed per unit of time or rather says the total amount of work done in a unit of time.

### **Turnaround Time**

It is the amount of time taken to execute a particular process, i.e. The interval from the time of submission of the process to the time of completion of the process(Wall clock time).

### **Waiting Time**

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

### **Load Average**

It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

### **Response Time**

Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

## **Preemptive Scheduling?**

Preemptive Scheduling is a scheduling method where 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.

At that time, the lower priority task holds for some time and resumes when the higher priority task finishes its execution.

## **Non- Preemptive Scheduling?**

Non-preemptive scheduling is a method that may be used when a process terminates or switches from a running to a waiting state. When processors are assigned to a process, they keep the process until it is eliminated or reaches a waiting state. When the processor starts the process execution, it must complete it before executing the other process, and it may not be interrupted in the middle.



## Preemptive Scheduling

A processor can be preempted to execute the different processes in the middle of any current process execution.

CPU utilization is more efficient compared to Non-Preemptive Scheduling.

Waiting and response time of preemptive Scheduling is less.

Preemptive Scheduling is prioritized. The highest priority process is a process that is currently utilized.

Preemptive Scheduling is flexible.

Examples: – Shortest Remaining Time First, Round Robin, etc.

In this process, the CPU is allocated to the processes for a specific time period.

Preemptive algorithm has the overhead of switching the process from the ready state to the running state and vice-versa.

## Non-preemptive Scheduling

Once the processor starts its execution, it must finish it before executing the other. It can't be paused in the middle.

CPU utilization is less efficient compared to preemptive Scheduling.

Waiting and response time of the non-preemptive Scheduling method is higher.

When any process enters the state of running, the state of that process is never deleted from the scheduler until it finishes its job.

Non-preemptive Scheduling is rigid.

Examples: First Come First Serve, Shortest Job First, Priority Scheduling, etc.

In this process, CPU is allocated to the process until it terminates or switches to the waiting state.

Non-preemptive Scheduling has no such overhead of switching the process from running into the ready state.



### •**Advantages of Preemptive Scheduling**

- Preemptive scheduling method is more robust, approach so one process cannot monopolize the CPU
- Choice of running task reconsidered after each interruption.
- Each event cause interruption of running tasks
- The OS makes sure that CPU usage is the same by all running process.
- In this, the usage of CPU is the same, i.e., all the running processes will make use of CPU equally.
- This scheduling method also improvises the average response time.
- Preemptive Scheduling is beneficial when we use it for the multi-programming environment.


### •**Advantages of Non-preemptive Scheduling**

- Offers low scheduling overhead
- Tends to offer high throughput
- It is conceptually very simple method
- Less computational resources need for Scheduling

### •**Disadvantages of Preemptive Scheduling**

- Need limited computational resources for Scheduling
- Takes a higher time by the scheduler to suspend the running task, switch the context, and dispatch the new incoming task.
- The process which has low priority needs to wait for a longer time if some high priority processes arrive continuously.

### **Disadvantages of Non-Preemptive Scheduling**

- It can lead to starvation especially for those real-time tasks
  - Bugs can cause a machine to freeze up
  - It can make real-time and priority Scheduling difficult
  - Poor response time for processes
- 

- FCFS Scheduling

- First come first serve** (FCFS) scheduling algorithm simply schedules the jobs according to their arrival time. The job which comes first in the ready queue will get the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

- Advantages of FCFS**

- Simple
- Easy
- First come, First serv

- Disadvantages of FCFS**

- The scheduling method is non preemptive, the process will run to the completion.
- Due to the non-preemptive nature of the algorithm, the problem of starvation may occur.
- Although it is easy to implement, but it is poor in performance since the average waiting time is higher as compare to other scheduling algorithms.

The Turnaround time and the waiting time are calculated by using the following formula.

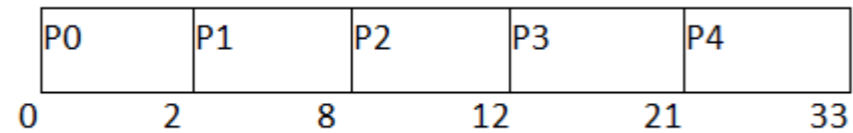
Turn Around Time = Completion Time - Arrival Time

Waiting Time = Turnaround time - Burst Time

The average waiting Time is determined by summing the respective waiting time of all the processes and divided the sum by the total number of processes.



Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
0	0	2	2	2	0
1	1	6	8	7	1
2	2	4	12	10	6
3	3	9	21	18	9
4	6	12	33	29	17

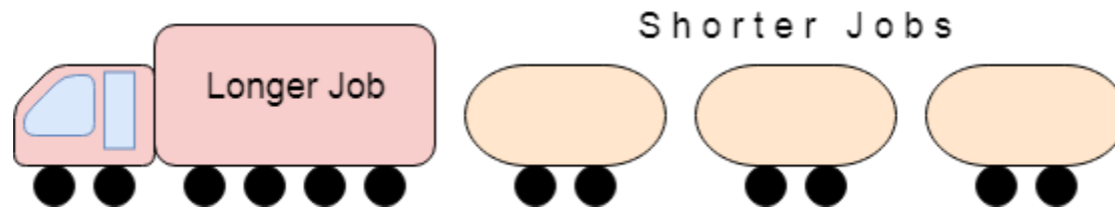


## Convoy Effect in FCFS

FCFS may suffer from the **convoy effect** if the burst time of the first job is the highest among all. As in the real life, if a convoy is passing through the road then the other persons may get blocked until it passes completely. This can be simulated in the Operating System also.

If the CPU gets the processes of the higher burst time at the front end of the ready queue then the processes of lower burst time may get blocked which means they may never get the CPU if the job in the execution has a very high burst time. This is called **convoy effect** or **starvation**.

The Convoy Effect, Visualized Starvation



## •Shortest Job First (SJF) Scheduling

SJF scheduling algorithm, schedules the processes according to their burst time.

- In SJF scheduling, the process with the lowest burst time, among the list of available processes in the ready queue, is going to be scheduled next.
- However, it is very difficult to predict the burst time needed for a process hence this algorithm is very difficult to implement in the system.

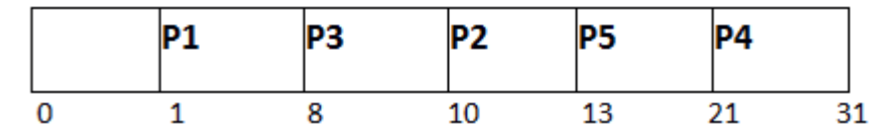
### •Advantages of SJF

- Maximum throughput
- Minimum average waiting and turnaround time

### •Disadvantages of SJF

- May suffer with the problem of starvation
- It is not implementable because the exact Burst time for a process can't be known in advance.

PID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	1	7	8	7	0
2	3	3	13	10	7
3	6	2	10	4	2
4	7	10	31	24	14
5	9	8	21	12	4



## Round Robin Scheduling Algorithm

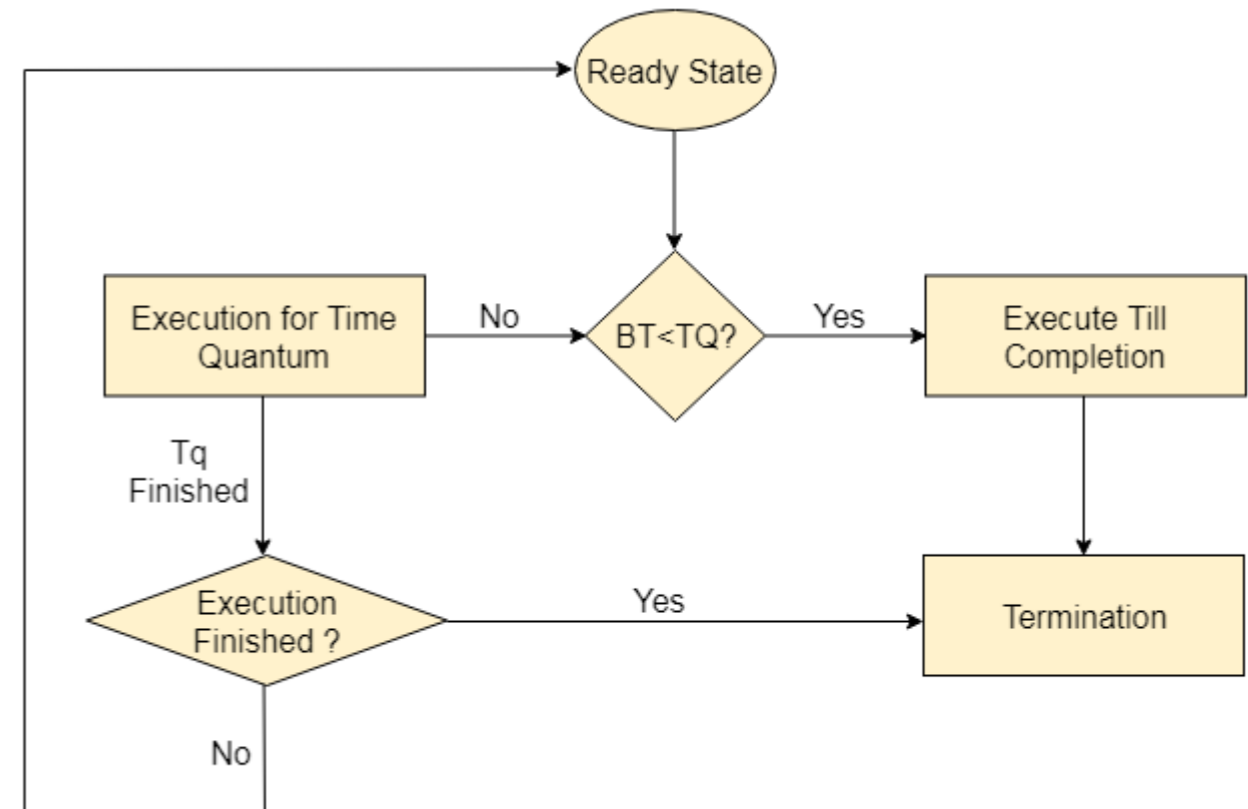
Round Robin scheduling algorithm is one of the most popular scheduling algorithm which can actually be implemented in most of the operating systems. This is the **preemptive version** of first come first serve scheduling. The Algorithm focuses on Time Sharing. In this algorithm, every process gets executed in a **cyclic way**. A certain time slice is defined in the system which is called time **quantum**. Each process present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will **terminate** else the process will go back to the **ready queue** and waits for the next turn to complete the execution.

### •Advantages

- It can be actually implementable in the system because it is not depending on the burst time.
- It doesn't suffer from the problem of starvation or convoy effect.
- All the jobs get a fair allocation of CPU.

### •Disadvantages

- The higher the time quantum, the higher the response time in the system.
- The lower the time quantum, the higher the context switching overhead in the system.
- Deciding a perfect time quantum is really a very difficult task in the system.



Process ID	Arrival Time	Burst Time
1	0	5
2	1	6
3	2	3
4	3	1
5	4	5
6	6	4

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	0	5	17	17	12
2	1	6	23	22	16
3	2	3	11	9	6
4	3	1	12	9	8
5	4	5	24	20	15
6	6	4	21	15	11

P1	P2	P3	P4	P5	P1	P6	P2	P5	
0	4	8	11	12	16	17	21	23	24



## Multiple Processor Scheduling

The availability of multiple processors makes scheduling more complicated, as there is more than one CPU that has to be kept busy at all times.

Load sharing is used in this case. It can be defined as balancing the load between multiple processors.

### **Approaches to Multiple processor scheduling:**

There are two approaches: Symmetric Multiprocessing and Asymmetric Multiprocessing.

**Symmetric Multiprocessing:** In Symmetric Multiprocessing, all processors are self-scheduling.

**Asymmetric Multiprocessing:** In Asymmetric Multiprocessing, scheduling decisions and I/O processes are handled by a single processor known as the Master Server.

### **Processor Affinity:**

In processor affinity, the processes have a priority for the processor which they are running.

**Soft affinity:** When the system tries to keep the processes on the same processor.

**Hard affinity:** When the process specifies that it should not be moved between the processors.

### **Load Balancing:**

Load Balancing is the phenomenon that keeps the workload evenly distributed across all processors in an SMP system so that one processor doesn't sit idle while the other is being overloaded.

**Push Migration:** A task regularly checks if there is an imbalance of load among the processors and then shifts\ distributes the load accordingly.

**Pull Migration:** It occurs when an idle processor pulls a task from an overloaded\ busy processor.



## Real Time Scheduling Algorithms

Real time scheduling is of two types: **Soft Real-Time scheduling** which does not guarantee when a critical real-time process will be scheduled; **Hard Real-Time scheduling** in which the process must be scheduled before the deadline.

### Points to Remember in Real Time Scheduling Algorithms

The foremost is that the processes are considered **periodic** i.e., the process will repeat itself after a fixed period of time. The period of a process is denoted by  $p$ . The next characteristic is the processing time  $t$  i.e., the time for which the process requires the CPU within each period. In other words processing time refers to the burst time. The final characteristic is the deadline  $d$ , i.e., the time before which the process must be serviced.

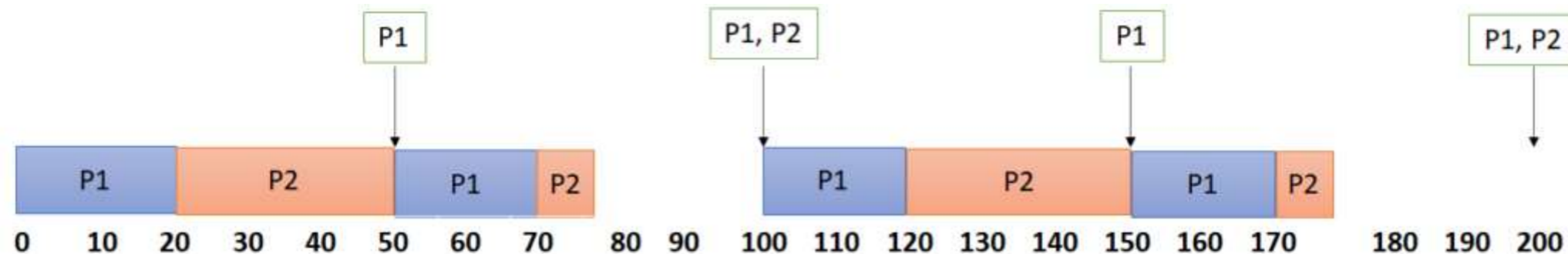
### Rate Monotonic Scheduling

In rate-monotonic scheduling algorithm a static priority policy with preemption is used. Whenever a high priority process arrives it will preempt a lower priority process. Every process gets the priority according to its period. Lower the period higher is the priority. Also, the processing time remains the same in each period.



Suppose there are two processes,  $P1$  and  $P2$ . The periods for  $P1$  and  $P2$  are 50 and 100. The processing times are  $t1 = 20$  for  $P1$  and  $t2 = 35$  for  $P2$ . The deadline for each process requires that it complete its CPU burst by the start of its next period.

Process	Period/Deadline	CPU Burst
P1	50	20
P2	100	35



### Failure of rate Monotonic Scheduling:

Assume that process  $P1$  has a period of  $p1 = 50$  and a CPU burst of  $t1 = 25$ .

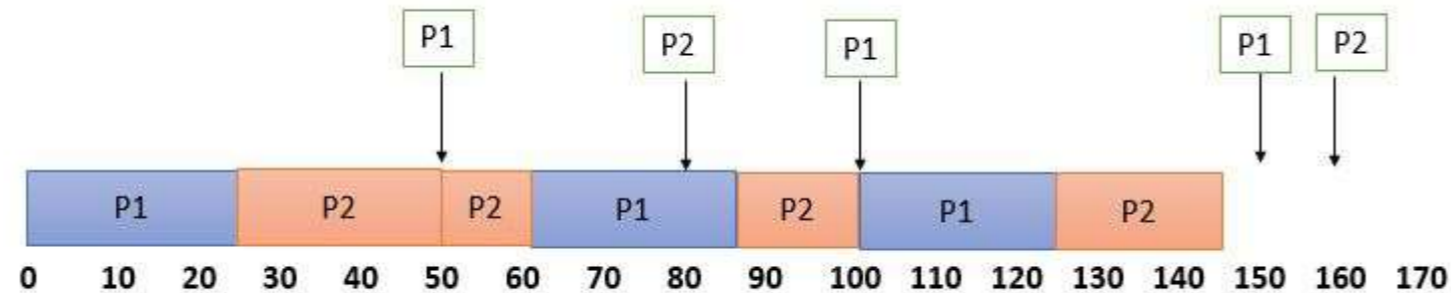
For  $P2$ , the corresponding values are  $p2 = 80$  and  $t2 = 35$



## Earliest Deadline First Scheduling

The second algorithm under real time scheduling is Earliest Deadline First Scheduling. This algorithm assigns priority to the process based on the deadline. Earlier the deadline, higher is the priority. Thus, the priorities keep on changing in this scheduling.

process	Period/Deadline	CPU Burst
P1	50	25
P2	80	35





**CHANDIGARH**  
**ENGINEERING COLLEGE**  
Building Careers. *Transforming Lives.*

Thank you

