UNIT-II

What is a process?

In computing, a process is **the instance of a computer program that is being executed by one or many threads**.

What is Process Scheduling?

Process Scheduling is the process of the process manager handling the removal of an active process from the CPU and selecting another process based on a specific strategy.

There are three types of process schedulers:

- Long term or Job Scheduler
- Short term or CPU Scheduler
- Medium-term Scheduler

Objectives of Process Scheduling Algorithm:

- Utilization of CPU at maximum level. **Keep CPU as busy as possible**.
- **Allocation of CPU should be fair**.
- **Throughput should be Maximum**. i.e. Number of processes that complete their execution per time unit should be maximized.
- **Minimum turnaround time**, i.e. time taken by a process to finish execution should be the least.
- There should be a **minimum waiting time** and the process should not starve in the ready queue.
- **Minimum response time.** It means that the time when a process produces the first response should be as less as possible.

Different terminologies to take care of in any CPU Scheduling algorithm

- **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

Things to take care while designing a CPU Scheduling algorithm?

The criteria include the following:

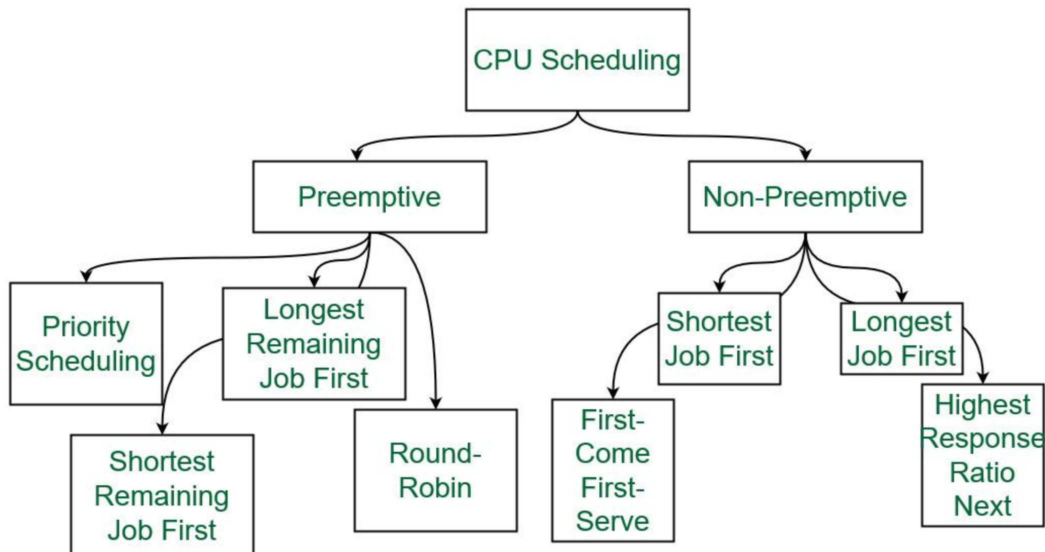
- **CPU utilization:** The main purpose of any CPU algorithm is to keep the CPU as busy as possible. Theoretically, CPU usage can range from 0 to 100 but in a real-time system, it varies from 40 to 90 percent depending on the system load.
- **Throughput:** The average CPU performance is the number of processes performed and completed during each unit. This is called throughput. The output may vary depending on the length or duration of the processes.
- **Turn round Time:** For a particular process, the important conditions are how long it takes to perform that process. The time elapsed from the time of process delivery to the time of completion is known as the conversion time. Conversion time is the amount of time spent waiting for memory access, waiting in line, using CPU, and waiting for I / O.
- **Waiting Time:** The Scheduling algorithm does not affect the time required to complete the process once it has started performing. It only affects the waiting time of the process i.e. the time spent in the waiting process in the ready queue.
- **Response Time:** In a collaborative system, turn around time is not the best option. The process may produce something early and continue to computing the new results while the previous results are released to the user. Therefore another method is the time taken in the submission of the application process until the first response is issued. This measure is called response time.

What are the different types of CPU Scheduling Algorithms?

There are mainly two types of scheduling methods:

- Preemptive Scheduling: Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to the ready state.
- Non-Preemptive Scheduling: Non-Preemptive scheduling is used when a process terminates , or when a process switches from running state to waiting state.

Different types of CPU Scheduling Algorithms



1. First Come First Serve:

FCFS considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.

Characteristics of FCFS:

- FCFS supports non-preemptive and **preemptive** CPU scheduling algorithms.
- Tasks are always executed on a First-come, First-serve concept.
- FCFS is easy to implement and use.
- This algorithm is not much efficient in performance, and the wait time is quite high.

Advantages of FCFS:

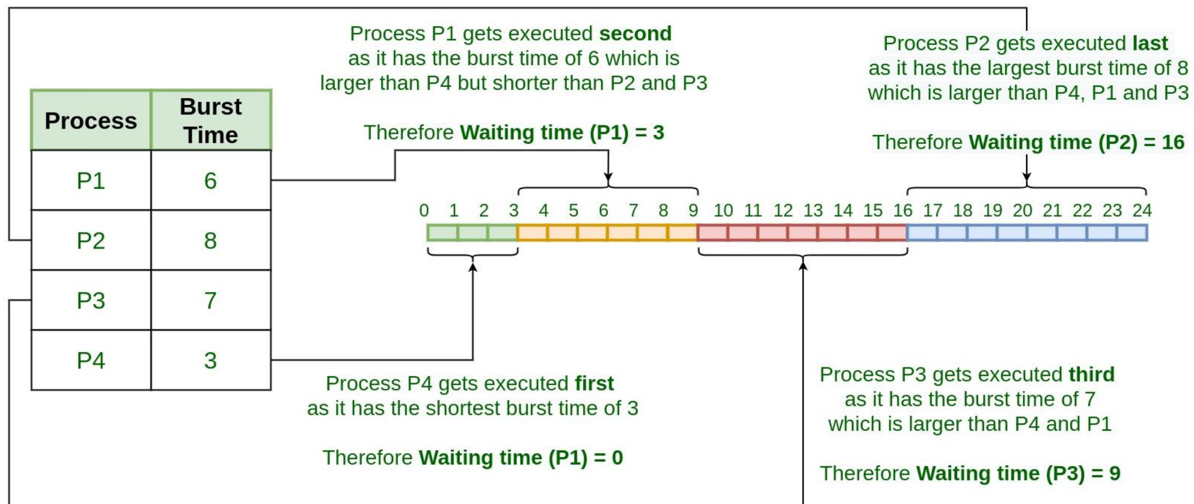
- Easy to implement
- First come, first serve method

Disadvantages of FCFS:

- FCFS suffers from **Convoy effect**.
- The average waiting time is much higher than the other algorithms.
- FCFS is very simple and **easy to implement and hence not much efficient**.

2. Shortest Job First(SJF):

Shortest job first (SJF) is a scheduling process that selects the waiting process with the smallest execution time to execute next. This scheduling method may or may not be preemptive. Significantly **reduces** the **average waiting time**for other processes waiting to be executed. The full form of SJF is Shortest Job First.



Characteristics of SJF:

- Shortest Job first has the advantage of having a minimum average waiting time among all operating system scheduling algorithms.
- It is associated with each task as a unit of time to complete.
- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.

Advantages of Shortest Job first:

- As SJF **reduces the average waiting time** thus, it is better than the first come first serve scheduling algorithm.
- SJF is generally used for **long term scheduling**

Disadvantages of SJF:

- One of the demerit SJF has is starvation.
- Many times it becomes complicated to predict the **length of the upcoming CPU** request

3. Longest Job First(LJF):

Longest Job First(LJF) scheduling process is just opposite of shortest job first (SJF), as the name suggests this algorithm is based upon the fact that the process with the largest burst time is processed first. **Longest Job First is non-preemptive** in nature.

Characteristics of LJF:

- Among all the processes waiting in a waiting queue, CPU is always assigned to the process having **largest burst time.**
- If **two processes have the same burst time** then the tie is broken using FCFS i.e. **the process that arrived first is processed first.**
- LJF CPU Scheduling can be of both **preemptive and non-preemptive types**.

Advantages of LJF:

- No other task can schedule until the longest job or process executes completely.
- All the jobs or processes finish at the same time approximately.

Disadvantages of LJF:

- Generally, the LJF algorithm gives a very high average waiting time and average turn-around time for a given set of processes.
- This may lead to convoy effect.

4. Priority Scheduling:

Preemptive Priority CPU Scheduling Algorithm is a pre-emptive method of CPU scheduling algorithm that works **based on the priority** of a process. In this algorithm, the editor sets the functions to be as important, meaning that the most important process must be done first. In the case of any conflict, that is, where there are more than one processor with equal value, then the most important CPU planning algorithm works on the basis of the FCFS (First Come First Serve) algorithm.

Characteristics of Priority Scheduling:

- Schedules tasks based on priority.
- When the higher priority work arrives while a task with less priority is executed, the higher priority work takes the place of the less priority one and
- The latter is suspended until the execution is complete.
- Lower is the number assigned, higher is the priority level of a process.

Advantages of Priority Scheduling:

- The average waiting time is less than FCFS
- Less complex

Disadvantages of Priority Scheduling:

- One of the most common demerits of the Preemptive priority CPU scheduling algorithm is the Starvation Problem. This is the problem in which a process has to wait for a longer amount of time to get scheduled into the CPU. This condition is called the starvation problem.

5. Round robin:

Round Robin is a CPU scheduling algorithm where each process is **cyclically assigned a fixed time slot**. It is the preemptive version of First come First Serve CPU Scheduling algorithm. Round Robin CPU Algorithm generally focuses on **Time Sharing technique.**

Characteristics of Round robin:

- It's simple, easy to use, and starvation-free as all processes get the balanced CPU allocation.
- One of the most widely used methods in CPU scheduling as a core.
- It is considered preemptive as the processes are given to the CPU for a very limited time.

Advantages of Round robin:

- Round robin seems to be fair as every process gets an equal share of CPU.
- The newly created process is added to the end of the ready queue.

6. Shortest Remaining Time First:

Shortest remaining time first is the preemptive version of the Shortest job first which we have discussed earlier where the processor is allocated to the job closest to completion. In SRTF the process with the smallest amount of time remaining until completion is selected to execute.

Characteristics of Shortest remaining time first:

- SRTF algorithm makes the processing of the jobs faster than SJF algorithm, given it's overhead charges are not counted.
- The context switch is done a lot more times in SRTF than in SJF and consumes the CPU's valuable time for processing. This adds up to its processing time and diminishes its advantage of fast processing.

Advantages of SRTF:

- In SRTF the short processes are handled very fast.
- The system also requires very little overhead since it only makes a decision when a process completes or a new process is added.

Disadvantages of SRTF:

- Like the shortest job first, it also has the potential for process starvation.
- Long processes may be held off indefinitely if short processes are continually added.

7. Longest Remaining Time First:

The longest remaining time first is a preemptive version of the longest job first scheduling algorithm. This scheduling algorithm is used by the operating system to program incoming processes for use in a systematic way. This algorithm schedules those processes first which have the longest processing time remaining for completion.

Characteristics of longest remaining time first:

- Among all the processes waiting in a waiting queue, the CPU is always assigned to the process having the largest burst time.
- If two processes have the same burst time then the tie is broken using FCFS i.e. the process that arrived first is processed first.
- LJF CPU Scheduling can be of both preemptive and non-preemptive types.

Advantages of LRTF:

- No other process can execute until the longest task executes completely.
- All the jobs or processes finish at the same time approximately.

Disadvantages of LRTF:

- This algorithm gives a very high average waiting time and average turn-around time for a given set of processes.
- This may lead to a convoy effect.

8. Highest Response Ratio Next:

Highest Response Ratio Next is a non-preemptive CPU Scheduling algorithm and it is considered as one of the most optimal scheduling algorithms. The name itself states that we need to find the response ratio of all available processes and select the one with the highest Response Ratio. A process once selected will run till completion.

Characteristics of Highest Response Ratio Next:

- The **criteria** for HRRN is **Response Ratio,** and the **mode** is **Non-Preemptive.**
- HRRN is considered as the modification of Shortest Job First to reduce the problem of starvation.
- In comparison with SJF, during the HRRN scheduling algorithm, the CPU is allotted to the next process which has the **highest response ratio** and not to the process having less burst time.

Response Ratio = (W + S)/S

Here, ***W*** *is the waiting time of the process so far and* ***S*** *is the Burst time of the process.*

Advantages of HRRN:

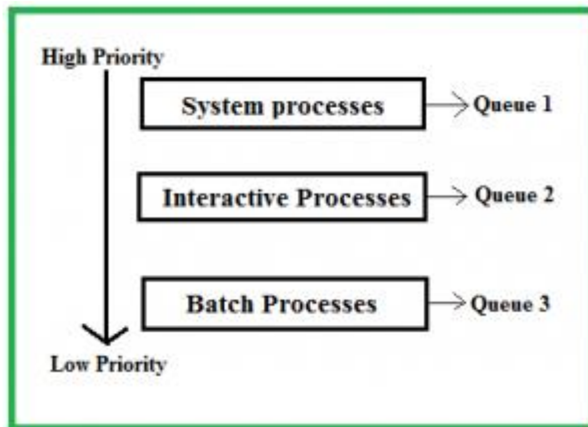
- HRRN Scheduling algorithm generally gives better performance than the shortest job first Scheduling.
- There is a reduction in waiting time for longer jobs and also it encourages shorter jobs.

Disadvantages of HRRN:

- The implementation of HRRN scheduling is not possible as it is not possible to know the burst time of every job in advance.
- In this scheduling, there may occur an overload on the CPU.

9. Multiple Queue Scheduling:

Processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a **foreground (interactive)** process and a **background (batch)** process. These two classes have different scheduling needs. For this kind of situation **Multilevel Queue Scheduling** is used.



The description of the processes in the above diagram is as follows:

- **System Processes:** The CPU itself has its process to run, generally termed as System Process.
- **Interactive Processes:** An Interactive Process is a type of process in which there should be the same type of interaction.
- **Batch Processes:** Batch processing is generally a technique in the Operating system that collects the programs and data together in the form of a **batch** before the **processing** starts.

Advantages of multilevel queue scheduling:

- The main merit of the multilevel queue is that it has a low scheduling overhead.

Disadvantages of multilevel queue scheduling:

- Starvation problem
- It is inflexible in nature

10. Multilevel Feedback Queue Scheduling:

Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling is like **Multilevel Queue Scheduling** but in this process can move between the queues. And thus, much more efficient than multilevel queue scheduling.

Characteristics of Multilevel Feedback Queue Scheduling:

- In a multilevel queue-scheduling algorithm, processes are permanently assigned to a queue on entry to the system, and processes are not allowed to move between queues.

- As the processes are permanently assigned to the queue, this setup has the advantage of low scheduling overhead,
- But on the other hand disadvantage of being inflexible.

Advantages of Multilevel feedback queue scheduling:

- It is more flexible
- It allows different processes to move between different queues

Disadvantages of Multilevel feedback queue scheduling:

- It also produces CPU overheads
- It is the most complex algorithm.

Comparison between various CPU Scheduling algorithms

Here is a brief comparison between different CPU scheduling algorithms:

Algorithm	Allocation is	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance
FCFS	According to the arrival time of the processes, the CPU is allocated.	Simple and easy to implement	Large.	No	No	Slow performance
SJF	Based on the lowest CPU burst	More complex than FCFS	Smaller than FCFS	No	Yes	Minimum Average Waiting Time

Algorithm	Allocation is	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance
	time (BT).					
LJFS	Based on the highest CPU burst time (BT)	More complex than FCFS	Depending on some measures e.g., arrival time, process size, etc.	No	Yes	Big turn-around time
LRTF	Same as LJFS the allocation of the CPU is based on the highest CPU burst time (BT). But it is preempti	More complex than FCFS	Depending on some measures e.g., arrival time, process size, etc.	Yes	Yes	The preference is given to the longer jobs

Algorithm	Allocation is	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance
	ve					
SRTF	Same as SJF the allocation of the CPU is based on the lowest CPU burst time (BT). But it is preemptive.	More complex than FCFS	Depending on some measures e.g., arrival time, process size, etc	Yes	Yes	The preference is given to the short jobs
RR	According to the order of the process arrives with fixed time	The complexity depends on Time Quantum size	Large as compared to SJF and Priority scheduling.	Yes	No	Each process has given a fairly fixed time

Algorithm	Allocation is	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance
	quantum (TQ)					
Priority Pre-emptive	According to the priority. The bigger priority task executes first	This type is less complex	Smaller than FCFS	Yes	Yes	Well performance but contain a starvation problem
Priority non-preemptive	According to the priority with monitoring the new incoming higher priority jobs	This type is less complex than Priority preemptive	Preemptive Smaller than FCFS	No	Yes	Most beneficial with batch systems
MLQ	According to the process	More complex than the	Smaller than FCFS	No	Yes	Good performance but

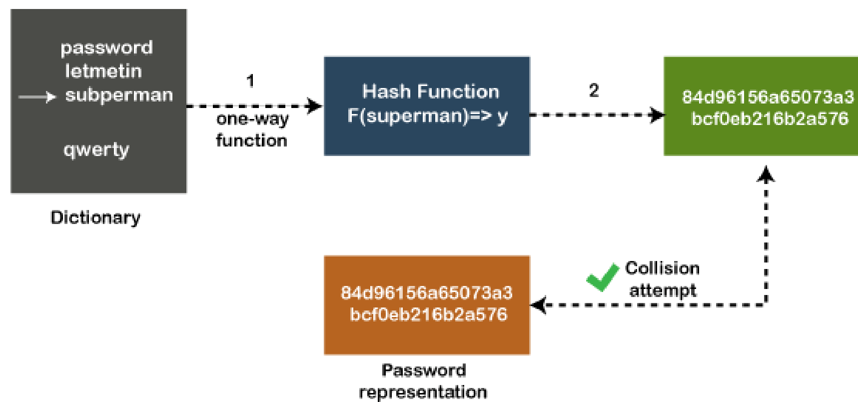
Algorithm	Allocation is	Complexity	Average waiting time (AWT)	Preemption	Starvation	Performance
	that resides in the bigger queue priority	priority scheduling algorithms				contain a starvation problem
MFLQ	According to the process of a bigger priority queue.	It is the most Complex but its complexity rate depends on the TQ size	Smaller than all scheduling types in many cases	No		

Inter Process Communication

The main aim or goal of this mechanism is to provide communications in between several processes (ie) the intercommunication allows a **process letting another process know that some event has occurred.**

Definition

"Inter-process communication is used for exchanging useful information between numerous threads in one or more processes (or programs)."



Role of Synchronization in Inter Process Communication

It is one of the essential parts of inter process communication. Typically, this is provided by interprocess communication control mechanisms, but sometimes it can also be controlled by communication processes.

These are the following methods that used to provide the synchronization:

1. **Mutual Exclusion**
2. **Semaphore**
3. **Barrier**
4. **Spinlock**

Mutual Exclusion:-

It is generally required that only one process thread can enter the critical section at a time. This also helps in synchronization and creates a stable state to avoid the race condition.

Semaphore:-

Semaphore is a type of variable that usually controls the access to the shared resources by several processes. Semaphore is further divided into two types which are as follows:

1. Binary Semaphore
2. Counting Semaphore

Barrier:-

A barrier typically not allows an individual process to proceed unless all the processes does not reach it. It is used by many parallel languages, and collective routines impose barriers.

Spinlock:-

Spinlock is a type of lock as its name implies. The processes are trying to acquire the spinlock waits or stays in a loop while checking that the lock is available or not. It is known as busy waiting because even though the process active, the process does not perform any functional operation (or task).

Approaches to Interprocess Communication

We will now discuss some different approaches to inter-process communication which are as follows:

These are a few different approaches for Inter- Process Communication:

1. **Pipes**
2. **Shared Memory**
3. **Message Queue**
4. **Direct Communication**
5. **Indirect communication**
6. **Message Passing**
7. **FIFO**

To understand them in more detail, we will discuss each of them individually.

Pipe:-

The pipe is a type of data channel that is unidirectional in nature. It means that the data in this type of data channel can be moved in only a single direction at a time. Still, one can use two-channel of this type, so that he can able to send and receive data in two processes. Typically, it uses the standard methods for input and output. These pipes are used in all types of POSIX systems and in different versions of window operating systems as well.

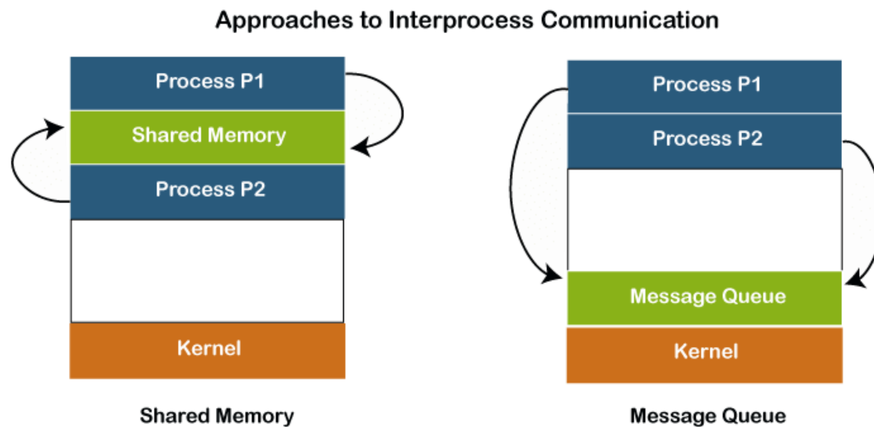
Shared Memory:-

It can be referred to as a type of memory that can be used or accessed by multiple processes simultaneously. It is primarily used so that the processes can communicate with each other.

Therefore the shared memory is used by almost all POSIX and Windows operating systems as well.

Message Queue:-

In general, several different messages are allowed to read and write the data to the message queue. In the message queue, the messages are stored or stay in the queue unless their recipients retrieve them. In short, we can also say that the message queue is very helpful in inter-process communication and used by all operating systems.



Message Passing:-

It is a type of mechanism that allows processes to synchronize and communicate with each other. However, by using the message passing, the processes can communicate with each other without restoring the hared variables.

Usually, the inter-process communication mechanism provides two operations that are as follows:

- send (message)
- received (message)

Note: The size of the message can be fixed or variable.

Direct Communication:-

In this type of communication process, usually, a link is created or established between two communicating processes. However, in every pair of communicating processes, only one link can exist.

Indirect Communication

Indirect communication can only exist or be established when processes share a common mailbox, and each pair of these processes shares multiple communication links. These shared links can be unidirectional or bi-directional.

FIFO:-

It is a type of general communication between two unrelated processes. It can also be considered as full-duplex, which means that one process can communicate with another process and vice versa.

Some other different approaches

- **Socket:-**

It acts as a type of endpoint for receiving or sending the data in a network. It is correct for data sent between processes on the same computer or data sent between different computers on the same network. Hence, it used by several types of operating systems.

- **File:-**

A file is a type of data record or a document stored on the disk and can be acquired on demand by the file server. Another most important thing is that several processes can access that file as required or needed.

- **Signal:-**

As its name implies, they are a type of signal used in inter process communication in a minimal way. Typically, they are the massages of systems that are sent by one process to another. Therefore, they are not used for sending data but for remote commands between multiple processes.

Usually, they are not used to send the data but to remote commands in between several processes.

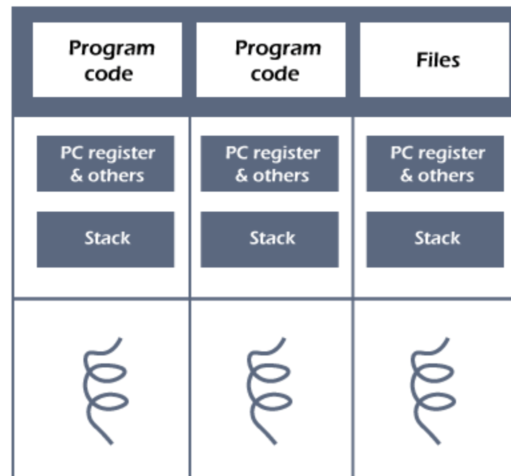
Why we need interprocess communication?

There are numerous reasons to use inter-process communication for sharing the data. Here are some of the most important reasons that are given below:

- It helps to speedup modularity
- Computational
- Privilege separation
- Convenience
- Helps operating system to communicate with each other and synchronize their actions as well.

Threads in Operating System (OS)

A thread is a single sequential flow of execution of tasks of a process so it is also known as thread of execution or thread of control. There is a way of thread execution inside the process of any operating system. Apart from this, there can be more than one thread inside a process. Each thread of the same process makes use of a separate program counter and a stack of activation records and control blocks. Thread is often referred to as a lightweight process.



Three threads of same process

The process can be split down into so many threads. **For example**, in a browser, many tabs can be viewed as threads. MS Word uses many threads - formatting text from one thread, processing input from another thread, etc.

Need of Thread:

- It takes far less time to create a new thread in an existing process than to create a new process.
- Threads can share the common data, they do not need to use Inter- Process communication.
- Context switching is faster when working with threads.
- It takes less time to terminate a thread than a process.

Types of Threads

In the operating system, there are two types of threads.

1. Kernel level thread.

2. User-level thread.

User-level thread

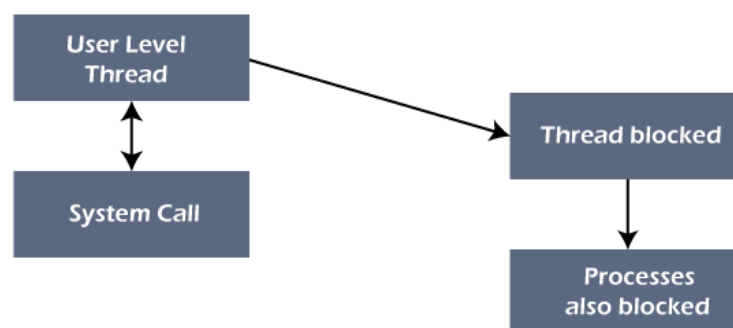
The operating system does not recognize the user-level thread. User threads can be easily implemented and it is implemented by the user. If a user performs a user-level thread blocking operation, the whole process is blocked. The kernel level thread does not know nothing about the user level thread. The kernel-level thread manages user-level threads as if they are single-threaded processes?examples: Java thread, POSIX threads, etc.

Advantages of User-level threads

1. The user threads can be easily implemented than the kernel thread.
2. User-level threads can be applied to such types of operating systems that do not support threads at the kernel-level.
3. It is faster and efficient.
4. Context switch time is shorter than the kernel-level threads.
5. It does not require modifications of the operating system.
6. User-level threads representation is very simple. The register, PC, stack, and mini thread control blocks are stored in the address space of the user-level process.
7. It is simple to create, switch, and synchronize threads without the intervention of the process.

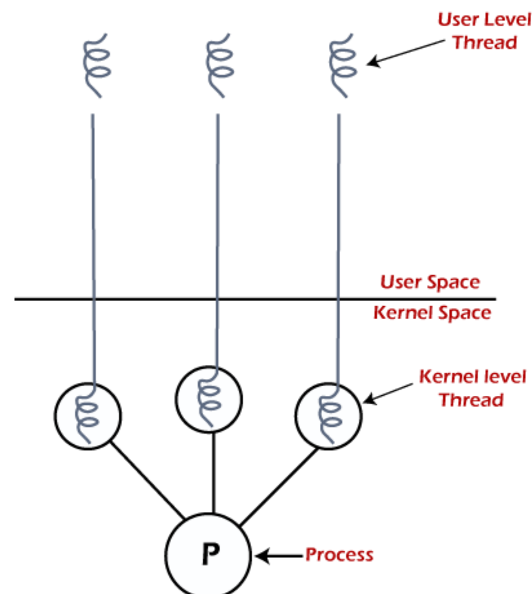
Disadvantages of User-level threads

1. User-level threads lack coordination between the thread and the kernel.
2. If a thread causes a page fault, the entire process is blocked.



Kernel level thread

The kernel thread recognizes the operating system. There is a thread control block and process control block in the system for each thread and process in the kernel-level thread. The kernel-level thread is implemented by the operating system. The kernel knows about all the threads and manages them. The kernel-level thread offers a system call to create and manage the threads from user-space. The implementation of kernel threads is more difficult than the user thread. Context switch time is longer in the kernel thread. If a kernel thread performs a blocking operation, the Banky thread execution can continue. Example: Window Solaris.



Advantages of Kernel-level threads

1. The kernel-level thread is fully aware of all threads.
2. The scheduler may decide to spend more CPU time in the process of threads being large numerical.
3. The kernel-level thread is good for those applications that block the frequency.

Disadvantages of Kernel-level threads

1. The kernel thread manages and schedules all threads.
2. The implementation of kernel threads is difficult than the user thread.
3. The kernel-level thread is slower than user-level threads.

Components of Threads

Any thread has the following components.

1. Program counter
2. Register set
3. Stack space

Benefits of Threads

- **Enhanced throughput of the system:** When the process is split into many threads, and each thread is treated as a job, the number of jobs done in the unit time increases. That is why the throughput of the system also increases.
- **Effective Utilization of Multiprocessor system:** When you have more than one thread in one process, you can schedule more than one thread in more than one processor.
- **Faster context switch:** The context switching period between threads is less than the process context switching. The process context switch means more overhead for the CPU.
- **Responsiveness:** When the process is split into several threads, and when a thread completes its execution, that process can be responded to as soon as possible.
- **Communication:** Multiple-thread communication is simple because the threads share the same address space, while in process, we adopt just a few exclusive communication strategies for communication between two processes.
- **Resource sharing:** Resources can be shared between all threads within a process, such as code, data, and files. Note: The stack and register cannot be shared between threads. There is a stack and register for each thread.

Multicore System

A single computing component with multiple cores (independent processing units) is known as a multicore processor. It denotes the presence of a single CPU with several cores in the system. Individually, these cores may read and run computer instructions. They work in such a way that the computer system appears to have several processors, although they are cores, not processors. These cores may execute normal processors instructions, including add, move data, and branch.

A single processor in a multicore system may run many instructions simultaneously, increasing the overall speed of the system's program execution. It decreases the amount of heat generated by the CPU while enhancing the speed with which instructions are executed. Multicore processors are used in various applications, including general-purpose, embedded, network, and graphics processing (GPU).

The software techniques used to implement the cores in a multicore system are responsible for the system's performance. The extra focus has been put on developing software that may execute in parallel because you want to achieve parallel execution with the help of many cores'

Advantages and disadvantages of Multicore System

There are various advantages and disadvantages of the multicore system. Some advantages and disadvantages of the multicore system are as follows:

Advantages

There are various advantages of the multicore system. Some advantages of the multicore system are as follows:

1. Multicore processors may execute more data than single-core processors.
2. When you are using multicore processors, the PCB requires less space.
3. It will have less traffic.
4. Multicores are often integrated into a single integrated circuit die or onto numerous dies but packaged as a single chip. As a result, Cache Coherency is increased.
5. These systems are energy efficient because they provide increased performance while using less energy.

Disadvantages

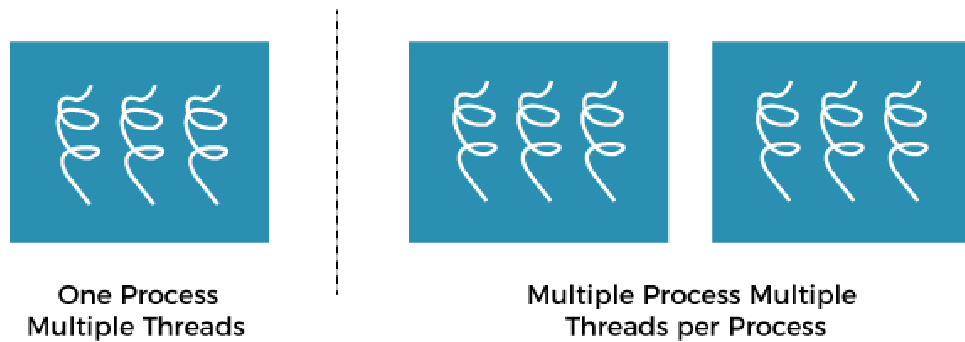
There are various disadvantages of the multicore system. Some disadvantages of the multicore system are as follows:

1. Some OSs are still using the single-core processor.
2. These are very difficult to manage than single-core processors.
3. These systems use huge electricity.
4. Multicore systems become hot while doing the work.
5. These are much expensive than single-core processors.
6. Operating systems designed for multicore processors will run slightly slower on single-core processors.

Multithreading Models

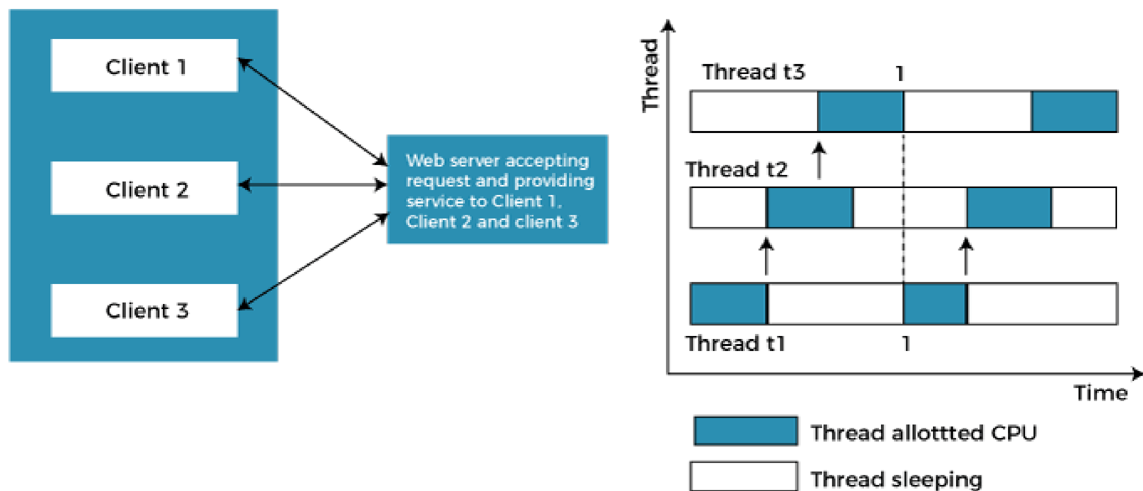
Multithreading Model:

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.

For example:



In the above example, client1, client2, and client3 are accessing the web server without any waiting. In multithreading, several tasks can run at the same time.

In an operating system, threads are divided into the user-level thread and the Kernel-level thread. User-level threads handled independent form above the kernel and thereby managed without any kernel support. On the opposite hand, the operating system directly manages the kernel-level threads. Nevertheless, there must be a form of relationship between user-level and kernel-level threads.

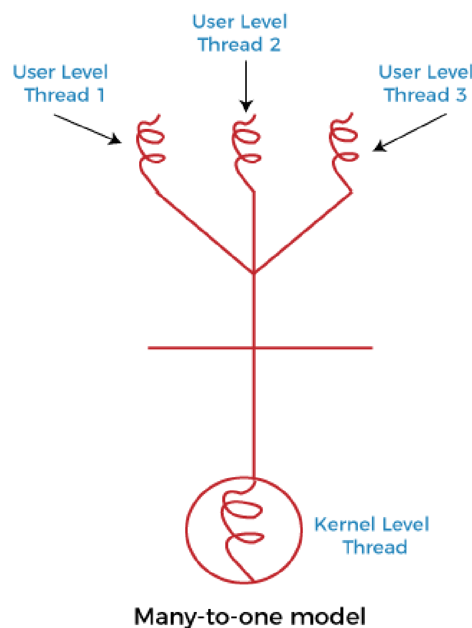
There exists three established multithreading models classifying these relationships are:

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

Many to one multithreading model:

The many to one model maps many user levels threads to one kernel thread. This type of relationship facilitates an effective context-switching environment, easily implemented even on the simple kernel with no thread support.

The disadvantage of this model is that since there is only one kernel-level thread schedule at any given time, this model cannot take advantage of the hardware acceleration offered by multithreaded processes or multi-processor systems. In this, all the thread management is done in the userspace. If blocking comes, this model blocks the whole system.

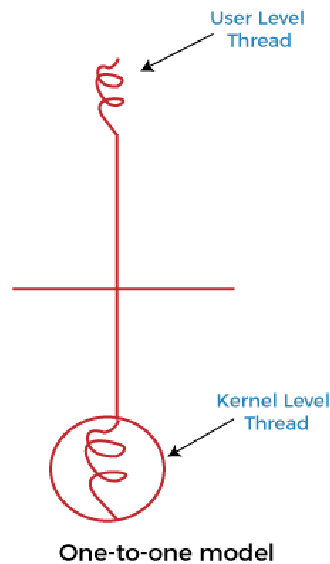


Many-to-one model

In the above figure, the many to one model associates all user-level threads to single kernel-level threads.

One to one multithreading model

The one-to-one model maps a single user-level thread to a single kernel-level thread. This type of relationship facilitates the running of multiple threads in parallel. However, this benefit comes with its drawback. The generation of every new user thread must include creating a corresponding kernel thread causing an overhead, which can hinder the performance of the parent process. Windows series and Linux operating systems try to tackle this problem by limiting the growth of the thread count.

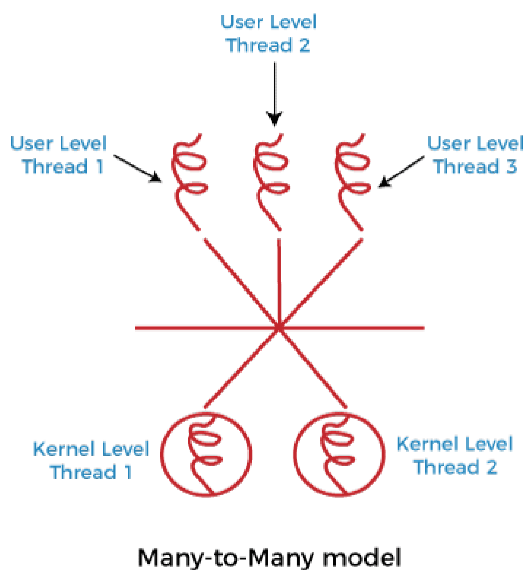


One-to-one model

In the above figure, one model associates that one user-level thread to a single kernel-level thread.

Many to Many Model multithreading model

In this type of model, there are several user-level threads and several kernel-level threads. The number of kernel threads created depends upon a particular application. The developer can create as many threads at both levels but may not be the same. The many to many model is a compromise between the other two models. In this model, if any thread makes a blocking system call, the kernel can schedule another thread for execution. Also, with the introduction of multiple threads, complexity is not present as in the previous models. Though this model allows the creation of multiple kernel threads, true concurrency cannot be achieved by this model. This is because the kernel can schedule only one process at a time.



Many-to-Many model

Process Synchronization: Critical-Section Problem

Process Synchronization is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.

It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time.
This can lead to the inconsistency of shared data. So the change made by one process not necessarily reflected when other processes accessed the same shared data. To avoid this type of inconsistency of data, the processes need to be synchronized with each other.

How Process Synchronization Works?

For Example, process A changing the data in a memory location while another process B is trying to read the data from the **same** memory location. There is a high probability that data read by the second process will be erroneous.

Sections of a Program

Here, are four essential elements of the critical section:

- **Entry Section:** It is part of the process which decides the entry of a particular process.
- **Critical Section:** This part allows one process to enter and modify the shared variable.
- **Exit Section:** Exit section allows the other process that are waiting in the Entry Section, to enter into the Critical Sections. It also checks that a process that finished its execution should be removed through this Section.
- **Remainder Section:** All other parts of the Code, which is not in Critical, Entry, and Exit Section, are known as the Remainder Section.

What is Critical Section Problem?

A critical section is a segment of code which can be accessed by a signal process at a specific point of time. The section consists of shared data resources that required to be accessed by other processes.

- The entry to the critical section is handled by the wait() function, and it is represented as P().
- The exit from a critical section is controlled by the signal() function, represented as V().

In the critical section, only a single process can be executed. Other processes, waiting to execute their critical section, need to wait until the current process completes its execution.

Rules for Critical Section

The critical section need to must enforce all three rules:

- **Mutual Exclusion:** Mutual Exclusion is a special type of binary semaphore which is used for controlling access to the shared resource. It includes a priority inheritance mechanism to avoid extended priority inversion problems. Not more than one process can execute in its critical section at one time.
- **Progress:** This solution is used when no one is in the critical section, and someone wants in. Then those processes not in their reminder section should decide who should go in, in a finite time.
- **Bound Waiting:** When a process makes a request for getting into critical section, there is a specific limit about number of processes can get into their critical section. So, when the limit is reached, the system must allow request to the process to get into its critical section.

Solutions To The Critical Section

In Process Synchronization, critical section plays the main role so that the problem must be solved.

Here are some widely used methods to solve the critical section problem.

Peterson Solution

Peterson’s solution is widely used solution to critical section problems. This algorithm was developed by a computer scientist Peterson that’s why it is named as a Peterson’s solution.

In this solution, when a process is executing in a critical state, then the other process only executes the rest of the code, and the opposite can happen. This method also helps to make sure that only a single process runs in the critical section at a specific time.

Example

	FLAG
P1	False
P2	True
P3	True
.	
.	
Pn	False

```
PROCESS Pi
FLAG[i] = true
while( (turn != i) AND (CS is !free) ){ wait;
}
CRITICAL SECTION FLAG[i] = false
turn = j; //choose another process to go to CS
```

- Assume there are N processes (P1, P2, … PN) and every process at some point of time requires to enter the Critical Section
- A FLAG[] array of size N is maintained which is by default false. So, whenever a process requires to enter the critical section, it has to set its flag as true. For example, If Pi wants to enter it will set FLAG[i]=TRUE.
- Another variable called TURN indicates the process number which is currently wating to enter into the CS.
- The process which enters into the critical section while exiting would change the TURN to another number from the list of ready processes.
- Example: turn is 2 then P2 enters the Critical section and while exiting turn=3 and therefore P3 breaks out of wait loop.

Synchronization Hardware

Some times the problems of the Critical Section are also resolved by hardware. Some operating system offers a lock functionality where a Process acquires a lock when entering the Critical section and releases the lock after leaving it.

So when another process is trying to enter the critical section, it will not be able to enter as it is locked. It can only do so if it is free by acquiring the lock itself.

Mutex Locks

Synchronization hardware not simple method to implement for everyone, so strict software method known as Mutex Locks was also introduced.

In this approach, in the entry section of code, a LOCK is obtained over the critical resources used inside the critical section. In the exit section that lock is released.

Semaphore Solution

Semaphore is simply a variable that is non-negative and shared between threads. It is another algorithm or solution to the critical section problem. It is a signaling mechanism and a thread that is waiting on a semaphore, which can be signaled by another thread.

It uses two atomic operations, 1)wait, and 2) signal for the process synchronization.

Example

```
WAIT ( S ):
while ( S <= 0 );
S = S - 1;
SIGNAL ( S ):
S = S + 1;
```

Classic Problems of Synchronization

Semaphore can be used in other synchronization problems besides Mutual Exclusion.

Below are some of the classical problem depicting flaws of process synchronaization in systems where cooperating processes are present.

We will discuss the following three problems:

1. Bounded Buffer (Producer-Consumer) Problem
2. Dining Philosophers Problem
3. The Readers Writers Problem

Bounded Buffer Problem

Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.

- This problem is generalised in terms of the **Producer Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes.
- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.
- In this Producers mainly produces a product and consumers consume the product, but both can use of one of the containers each time.
- The main complexity of this problem is that we must have to maintain the count for both empty and full containers that are available.

Dining Philosophers Problem

- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the centre, When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

The Readers Writers Problem

- In this problem there are some processes(called **readers**) that only read the shared data, and never change it, and there are other

processes(called **writers**) who may change the data in addition to reading, or instead of reading it.

- There are various type of readers-writers problem, most centred on relative priorities of readers and writers.
- The main complexity with this problem occurs from allowing more than one reader to access the data at the same time.

CPU Scheduling

In the uniprogrammming systems like MS DOS, when a process waits for any I/O operation to be done, the CPU remains idol. This is an overhead since it wastes the time and causes the problem of starvation. However, In Multiprogramming systems, the CPU doesn't remain idle during the waiting time of the Process and it starts executing other processes. Operating System has to define which process the CPU will be given.

In Multiprogramming systems, the Operating system schedules the processes on the CPU to have the maximum utilization of it and this procedure is called **CPU scheduling**. The Operating System uses various scheduling algorithm to schedule the processes.

This is a task of the short term scheduler to schedule the CPU for the number of processes present in the Job Pool. Whenever the running process requests some IO operation then the short term scheduler saves the current context of the process (also called PCB) and changes its state from running to waiting. During the time, process is in waiting state; the Short term scheduler picks another process from the ready queue and assigns the CPU to this process. This procedure is called **context switching**.

What is saved in the Process Control Block?

The Operating system maintains a process control block during the lifetime of the process. The Process control block is deleted when the process is terminated or killed. There is the following information which is saved in the process control block and is changing with the state of the process.

Process ID
Process State
Pointer
Priority
Program Counter
CPU Registers
I/O Information
Accounting Information
etc.

Why do we need Scheduling?

In Multiprogramming, if the long term scheduler picks more I/O bound processes then most of the time, the CPU remains idol. The task of Operating system is to optimize the utilization of resources.

If most of the running processes change their state from running to waiting then there may always be a possibility of deadlock in the system. Hence to reduce this overhead, the OS needs to schedule the jobs to get the optimal utilization of CPU and to avoid the possibility to deadlock.

Scheduling Criteria

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

The criteria include the following:

1. **CPU utilization:** The main objective of any CPU scheduling algorithm is to keep the CPU as busy as possible. Theoretically, CPU utilization 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.
2. **Throughput:** A measure of the work done by the CPU is the number of processes being executed and completed per unit of time. This is called throughput. The throughput may vary depending on the length or duration of the processes.
3. **Turnaround time:** For a particular process, an important criterion 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 the ready queue, executing in CPU, and waiting for I/O. The formula to calculate Turn Around Time = Compilation Time – Arrival Time.
4. **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. The formula for calculating Waiting Time = Turnaround Time – Burst Time.

5. **Response time:** In an interactive system, turn-around time is not the best criterion. A process may produce some output fairly early and continue computing new results while previous results are being output to the user. Thus another criterion is the time taken from submission of the process of the request until the first response is produced. This measure is called response time. The formula to calculate Response Time = CPU Allocation Time(when the CPU was allocated for the first) – Arrival Time
6. **Completion time:** The completion time is the time when the process stops executing, which means that the process has completed its burst time and is completely executed.
7. **Priority:** If the operating system assigns priorities to processes, the scheduling mechanism should favor the higher-priority processes.
8. **Predictability:** A given process always should run in about the same amount of time under a similar system load.

There are various CPU Scheduling algorithms such as-

- First Come First Served (FCFS)
- Shortest Job First (SJF)
- Longest Job First (LJF)
- Priority Scheduling
- Round Robin (RR)
- Shortest Remaining Time First (SRTF)
- Longest Remaining Time First (LRTF)

Thread Scheduling

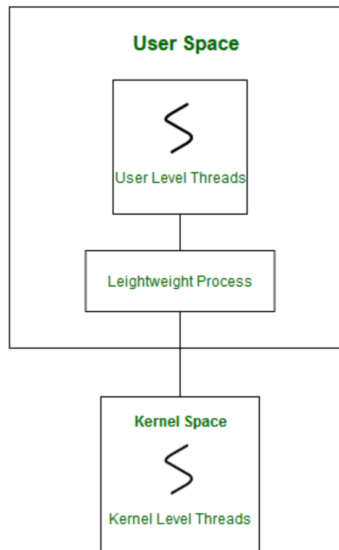
Scheduling of threads involves two boundary scheduling,

- Scheduling of user level threads (ULT) to kernel level threads (KLT) via lightweight process (LWP) by the application developer.
- Scheduling of kernel level threads by the system scheduler to perform different unique os functions.

Lightweight Process (LWP) :

Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources. Thread library schedules which thread of a process to run on which LWP and how long. The number of LWP created by the thread library depends on the type of application. In the case of an I/O bound application, the number of LWP depends on the

number of user-level threads. This is because when an LWP is blocked on an I/O operation, then to invoke the other ULT the thread library needs to create and schedule another LWP. Thus, in an I/O bound application, the number of LWP is equal to the number of the ULT. In the case of a CPU bound application, it depends only on the application. Each LWP is attached to a separate kernel-level thread.



In real-time, the first boundary of thread scheduling is beyond specifying the scheduling policy and the priority. It requires two controls to be specified for the User level threads: Contention scope, and Allocation domain. These are explained as following below.

1. Contention Scope :

The word contention here refers to the competition or fight among the User level threads to access the kernel resources. Thus, this control defines the extent to which contention takes place. It is defined by the application developer using the thread library. Depending upon the extent of contention it is classified as **Process Contention Scope** and **System Contention Scope**.

1. **Process Contention Scope (PCS) –**
   The contention takes place among threads **within a same process**. The thread library schedules the high-prioritized PCS thread to access the resources via available LWPs (priority as specified by the application developer during thread creation).
2. **System Contention Scope (SCS) –**
   The contention takes place among **all threads in the system**. In this case, every SCS thread is associated to each LWP by the thread library and are scheduled by the system scheduler to access the kernel resources. In LINUX and UNIX operating systems, the POSIX Pthread library provides a function *Pthread_attr_setscope* to define the type of contention scope for a thread during its creation.

```
int Pthread_attr_setscope(pthread_attr_t *attr, int scope)
```

1. The first parameter denotes to which thread within the process the scope is defined.
   The second parameter defines the scope of contention for the thread pointed. It takes two values.

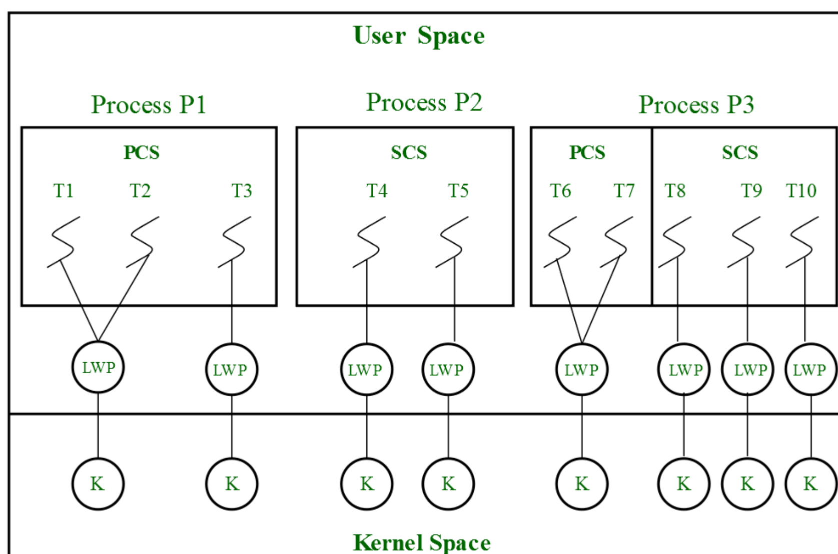
```
PTHREAD_SCOPE_SYSTEM
PTHREAD_SCOPE_PROCESS
```

1. If the scope value specified is not supported by the system, then the function returns *ENOTSUP*.

2. Allocation Domain :

The allocation domain is **a set of one or more resources** for which a thread is competing. In a multicore system, there may be one or more allocation domains where each consists of one or more cores. One ULT can be a part of one or more allocation domain. Due to this high complexity in dealing with hardware and software architectural interfaces, this control is not specified. But by default, the multicore system will have an interface that affects the allocation domain of a thread.

Consider a scenario, an operating system with three process P1, P2, P3 and 10 user level threads (T1 to T10) with a single allocation domain. 100% of CPU resources will be distributed among all the three processes. The amount of CPU resources allocated to each process and to each thread depends on the contention scope, scheduling policy and priority of each thread defined by the application developer using thread library and also depends on the system scheduler. These User level threads are of a different contention scope.



In this case, the contention for allocation domain takes place as follows,

1. **Process P1:**
   All PCS threads T1, T2, T3 of Process P1 will compete among themselves. The PCS threads of the same process can share one or more LWP. T1 and T2 share an LWP and T3 are allocated to a separate LWP. Between T1 and T2 allocation of kernel resources via LWP is based on preemptive priority scheduling by the thread library. A Thread with a high priority will preempt low priority threads. Whereas, thread T1 of process p1 cannot preempt thread T3 of process p3 even if the priority of T1 is greater than the priority of T3. If the priority is equal, then the allocation of ULT to available LWPs is based on the scheduling policy of threads by the system scheduler(not by thread library, in this case).

2. **Process P2:**
   Both SCS threads T4 and T5 of process P2 will compete with processes P1 as a whole and with SCS threads T8, T9, T10 of process P3. The system scheduler will schedule the kernel resources among P1, T4, T5, T8, T9, T10, and PCS threads (T6, T7) of process P3 considering each as a separate process. Here, the Thread library has no control of scheduling the ULT to the kernel resources.

3. **Process P3:**
   Combination of PCS and SCS threads. Consider if the system scheduler allocates 50% of CPU resources to process P3, then 25% of resources is for process scoped threads and the remaining 25% for system scoped threads. The PCS threads T6 and T7 will be allocated to access the 25% resources based on the priority by the thread library. The SCS threads T8, T9, T10 will divide the 25% resources among themselves and access the kernel resources via separate LWP and KLT. The SCS scheduling is by the system scheduler.

Note:
For every system call to access the kernel resources, a Kernel Level thread is created and associated to separate LWP by the system scheduler.

```
Number of Kernel Level Threads = Total Number of LWP
Total Number of LWP = Number of LWP for SCS + Number of LWP for PCS
Number of LWP for SCS = Number of SCS threads
Number of LWP for PCS = Depends on application developer
```

Here,

```
Number of SCS threads = 5
Number of LWP for PCS = 3
```

```
Number of SCS threads = 5
Number of LWP for SCS = 5
Total Number of LWP   = 8 (=5+3)
Number of Kernel Level Threads = 8
```

Advantages of PCS over SCS :

- If all threads are PCS, then context switching, synchronization, scheduling everything takes place within the userspace. This reduces system calls and achieves better performance.
- PCS is cheaper than SCS.
- PCS threads share one or more available LWPs. For every SCS thread, a separate LWP is associated.For every system call, a separate KLT is created.
- The number of KLT and LWPs created highly depends on the number of SCS threads created. This increases the kernel complexity of handling scheduling and synchronization. Thereby, results in a limitation over SCS thread creation, stating that, the number of SCS threads to be smaller than the number of PCS threads.
- If the system has more than one allocation domain, then scheduling and synchronization of resources becomes more tedious. Issues arise when an SCS thread is a part of more than one allocation domain, the system has to handle n number of interfaces.

The second boundary of thread scheduling involves CPU scheduling by the system scheduler. The scheduler considers each kernel-level thread as a separate process and provides access to the kernel resources.