Unit II – Process Management

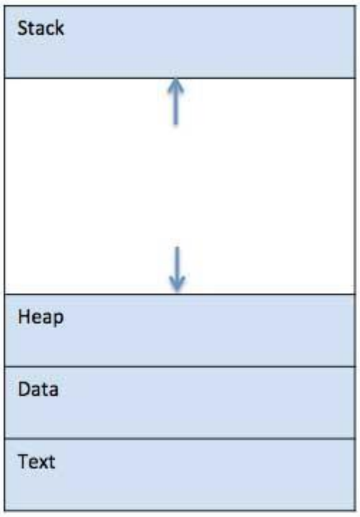
Definition:

A process is basically a program in execution or instance of the program execution. The execution of a process must progress in a sequential fashion.

- Process is not as same as program code but a lot more than it.
- A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity.
- Attributes held by process include hardware state, memory, CPU etc.

Process memory is divided into four sections for efficient working :

- The ***Text section*** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The ***Data section*** is made up the global and static variables, allocated and initialized prior to executing the main.
- The ***Heap*** is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The ***Stack*** is used for local variables. Space on the stack is reserved for local variables when they are declared.

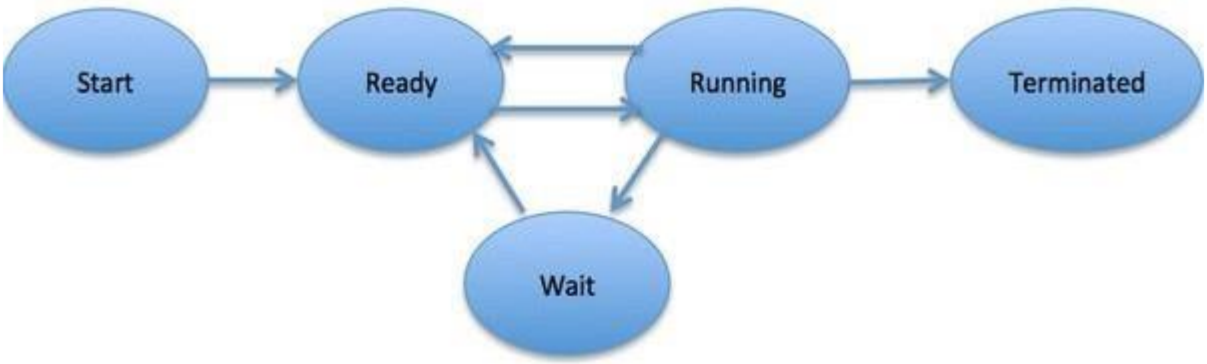


Process States:

When a process executes, it passes through different states. These stages may differ in different operating systems.

In general, a process can have one of the following five states at a time.

S.N.	State & Description
1	**Start:** This is the initial state when a process is first started/created.
2	**Ready:** 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 **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	**Running:** 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	**Waiting:** Process moves into 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	**Terminated or Exit:** 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 State Diagram

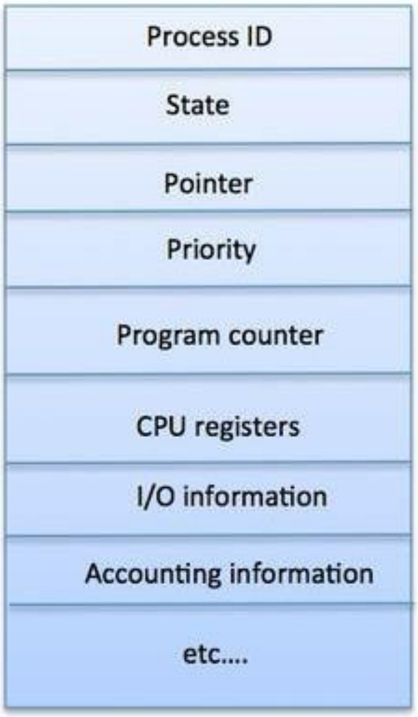
Process Control Block (PCB):

- 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 –

S.N.	Information & Description
1	**Process State:** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges:** This is required to allow/disallow access to system resources.
3	**Process ID:** Unique identification for each of the process in the operating system.
4	**Pointer:** A pointer to parent process.
5	**Program Counter: P**rogram Counter is a pointer to the address of the next instruction to be executed for this process.
6	**CPU registers:** Various CPU registers where process need to be stored for execution for running state.

7	**CPU Scheduling Information:** Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information:** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	**Accounting information:** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information:** This includes a list of I/O devices allocated to the process.

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −



Process Control Block (PCB) Diagram

The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

3.2 Process Scheduling

Definition

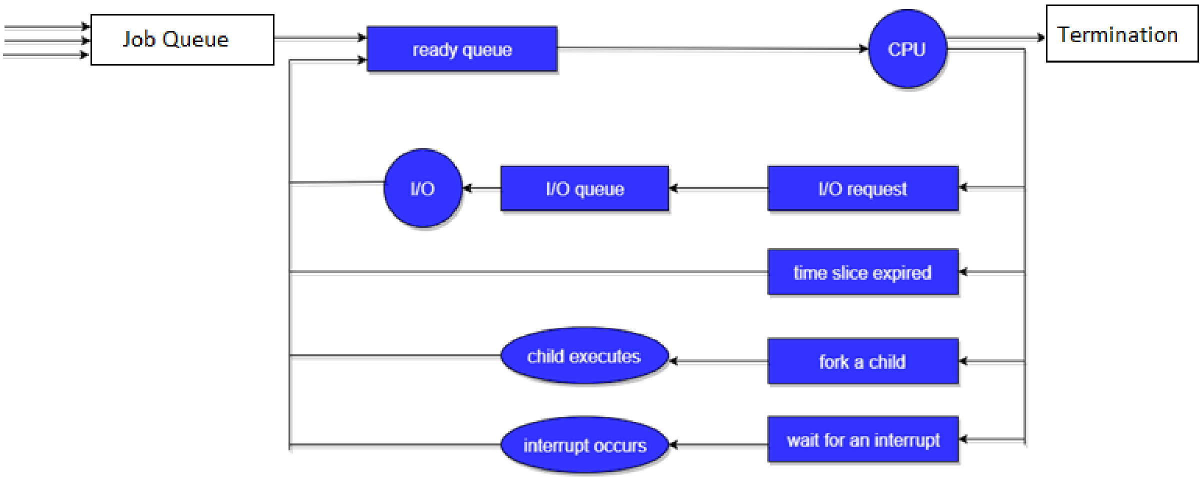
- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- Process scheduling is an essential part of a Multiprogramming operating systems.
- Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

What are Scheduling Queues?

- All processes, upon entering into the system, are stored in the **Job Queue**.
- Processes in the Ready state are placed in the **Ready Queue**.
- Processes waiting for a device to become available are placed in **Device Queues**. There are unique device queues available for each I/O device.

A new process is initially put in the **Ready queue**. It waits in the ready queue until it is selected for execution (or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

- The process could issue an I/O request, and then be placed in the **I/O queue**.
- The process could create a new sub-process and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.



In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Schedulers:

- Schedulers are special system software which handle process scheduling in various ways.
- Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −
  - Long-Term Scheduler
  - Short-Term Scheduler
  - Medium-Term Scheduler

Long Term Scheduler

- It is also called a job scheduler.
- A long-term scheduler determines which programs are admitted to the system for processing.
- It selects processes from the queue and loads them into memory for execution.
- Process loads into the memory for CPU scheduling.
- The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound.

- It also controls the degree of multiprogramming.
- If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- On some systems, the long-term scheduler may not be available or minimal.
- Time-sharing operating systems have no long term scheduler.
- When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler:

- It is also called as CPU scheduler.
- Its main objective is to increase system performance in accordance with the chosen set of criteria.
- It is the change of ready state to running state of the process.
- CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.
- Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

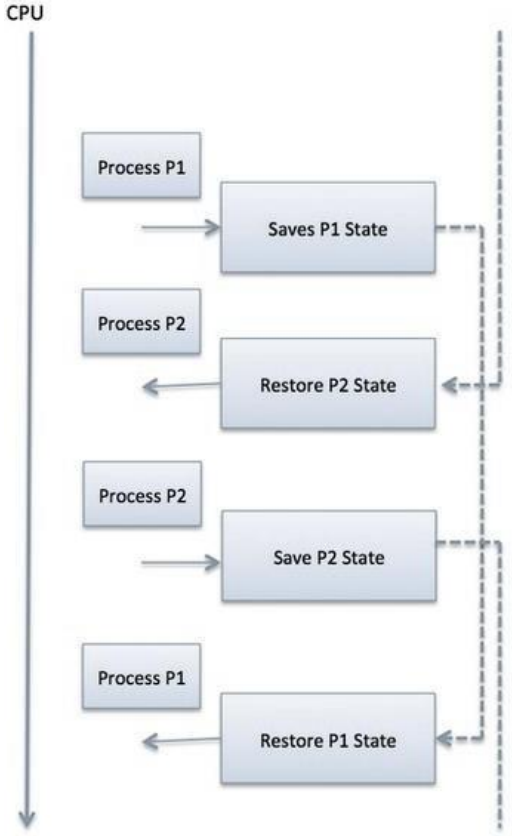
- Medium-term scheduling is a part of swapping.
- It removes the processes from the memory.
- It reduces the degree of multiprogramming.
- The medium-term scheduler is in-charge of handling the swapped out-processes.
- A running process may become suspended if it makes an I/O request.
- A suspended process cannot make any progress towards completion.
- In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage.
- This process is called swapping, and the process is said to be swapped out or rolled out.
- Swapping may be necessary to improve the process mix.

Comparison among Scheduler

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Context Switch:

- A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time.
- Using this technique, a context switcher enables multiple processes to share a single CPU.
- Context switching is an essential part of a multitasking operating system features.
- When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block.
- After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc.
- At that point, the second process can start executing.



- Context switches are computationally intensive since register and memory state must be saved and restored.
- To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers.
- When the process is switched, the following information is stored for later use.
    - Program Counter
    - Scheduling information
    - Base and limit register value
    - Currently used register
    - Changed State
    - I/O State information
    - Accounting information

3.3 Inter-process Communication

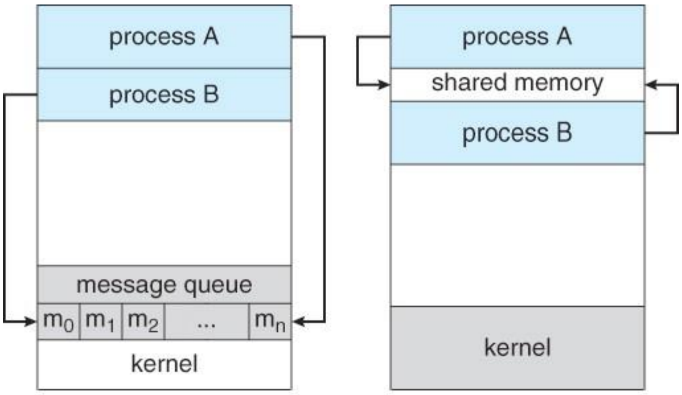
- Processes executing concurrently in the operating system might be either independent processes or cooperating processes.
- A process is independent if it cannot be affected by the other processes executing in the system**.**
- ***Inter Process Communication (IPC)*** is a mechanism that involves communication of one process with another process. This usually occurs only in one system.
- Communication can be of two types −
  - Between related processes initiating from only one process, such as parent and child processes.
  - Between unrelated processes, or two or more different processes.
- Processes can communicate with each other using these two ways:
  - Shared Memory
  - Message passing

Shared Memory

- Shared memory is the memory that can be simultaneously accessed by multiple processes.
- This is done so that the processes can communicate with each other.
- All POSIX systems, as well as Windows operating systems use shared memory.

Message Queue

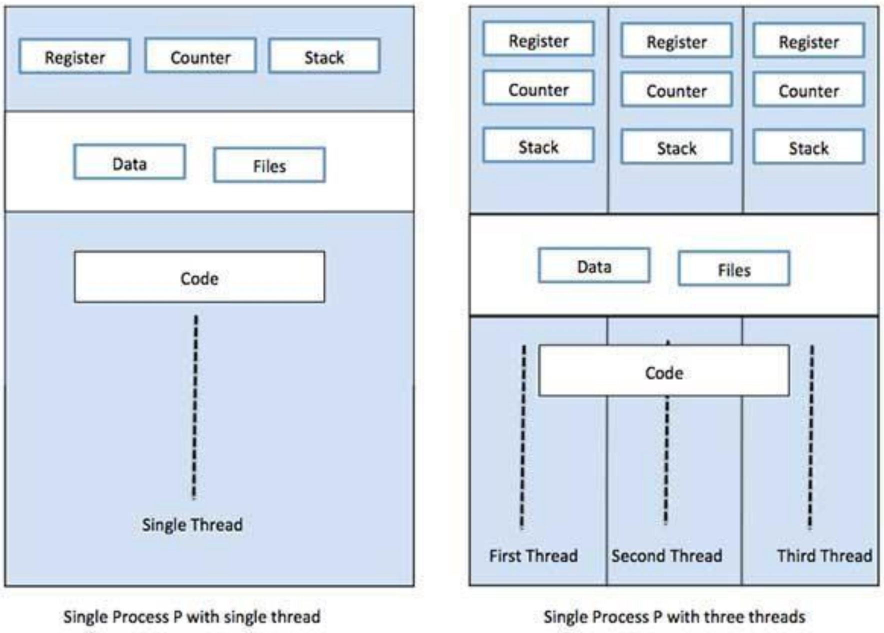
- Multiple processes can read and write data to the message queue without being connected to each other.
- Messages are stored in the queue until their recipient retrieves them.
- Message queues are quite useful for inter-process communication and are used by most operating systems.
- If two processes p1 and p2 want to communicate with each other, they proceed as follow:
  - Establish a communication link (if a link already exists, no need to establish it again.)
  - Start exchanging messages using basic primitives.
  - We need at least two primitives:
    send(message, destinaion) or send(message)
    receive(message, host) or receive(message)



(a) Message Passing (b) Shared Memory

3.4 What is Thread?

- A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.
- A thread shares with its peer threads few information like code segment, data segment and open files.
- When one thread alters a code segment memory item, all other threads see that.
- A thread is also called a **lightweight process**.
- Threads provide a way to improve application performance through parallelism.
- Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.
- Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control.
- Threads have been successfully used in implementing network servers and web server.
- They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.
- The following figure shows the working of a single-threaded and a multithreaded process.



Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.

4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

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

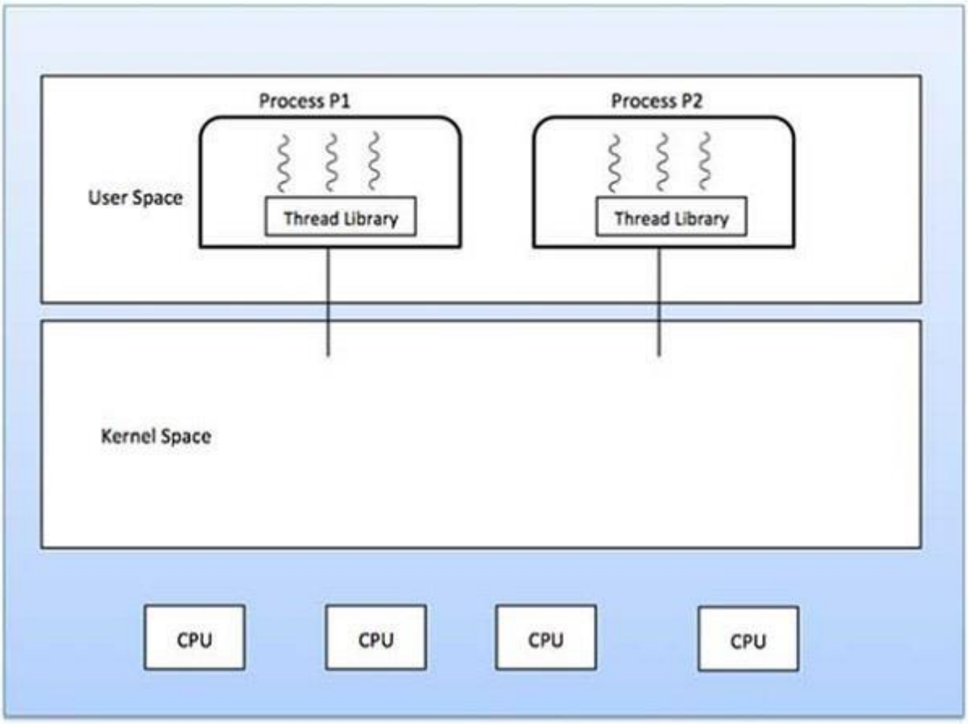
Types of Thread

Threads are implemented in following two ways −

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

User Level Threads

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



Advantages

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

Disadvantages

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

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

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

Disadvantages

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

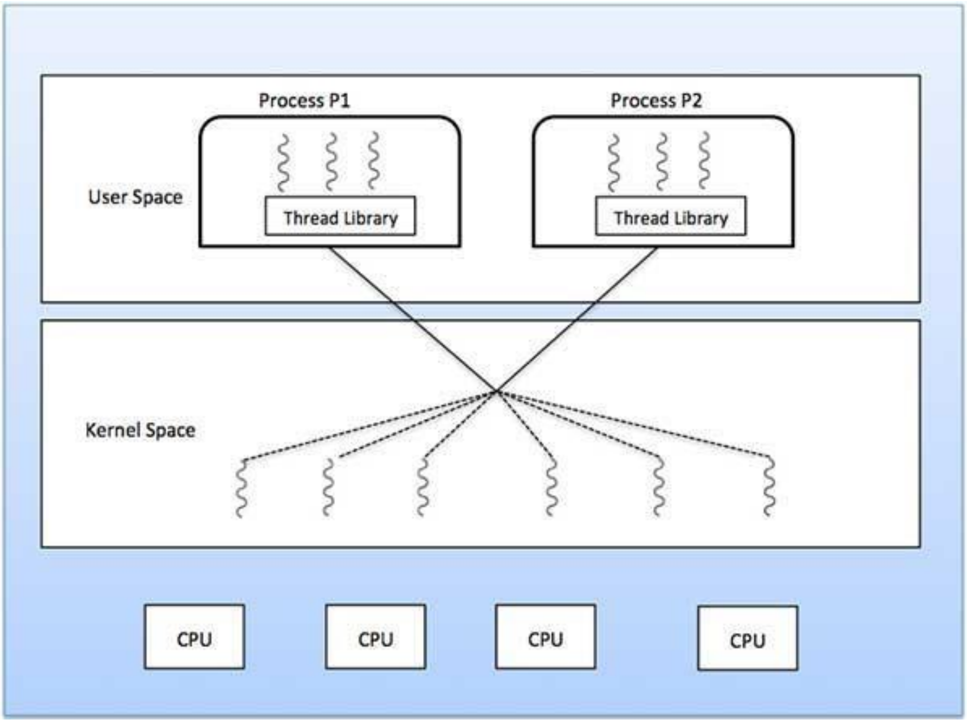
Multithreading Models

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

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

Many to Many Model

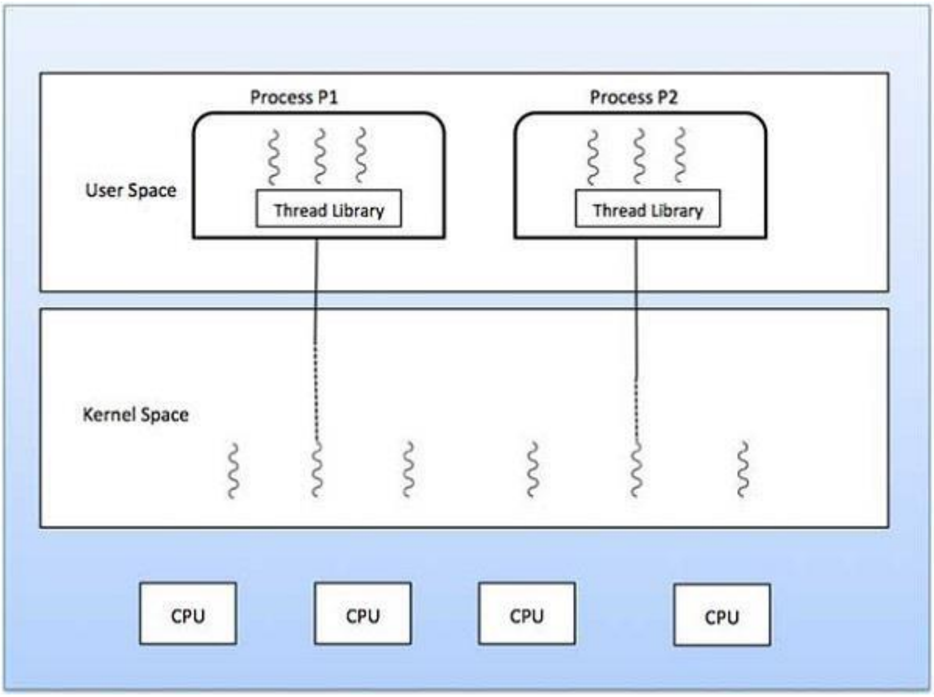
- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.
- The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads.
- In this model, developers can create as many user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



Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

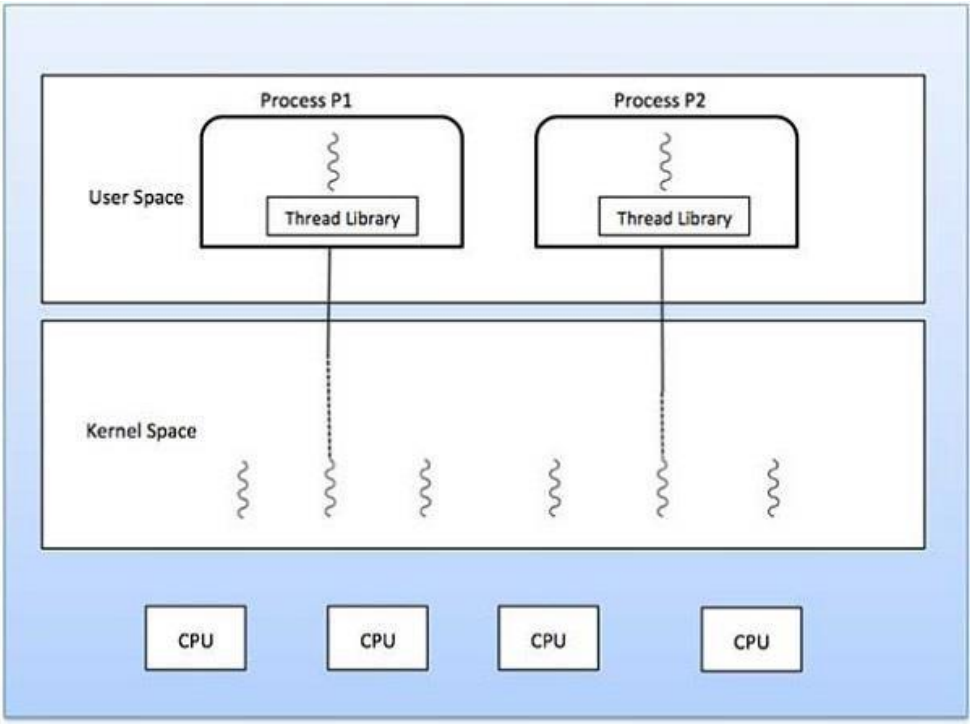
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



One to One Model

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



Difference between User-Level & Kernel-Level Thread

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