UNIT IV
Operating System - Multi-Threading

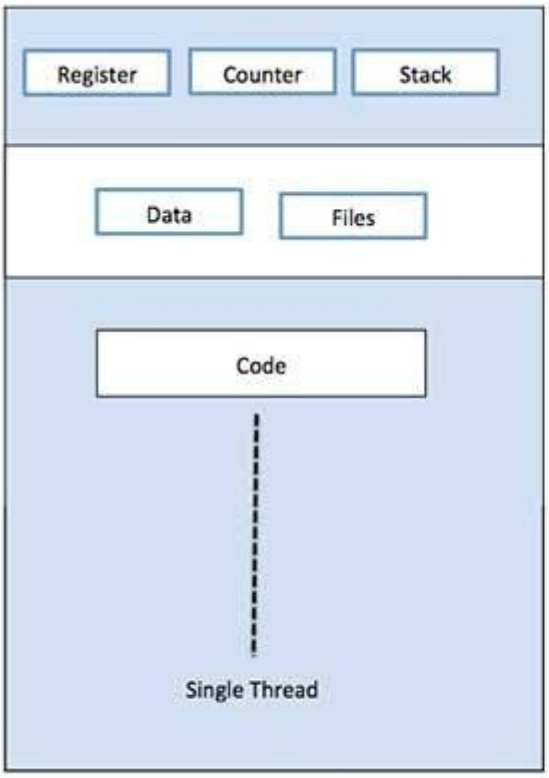
What is Thread?

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.

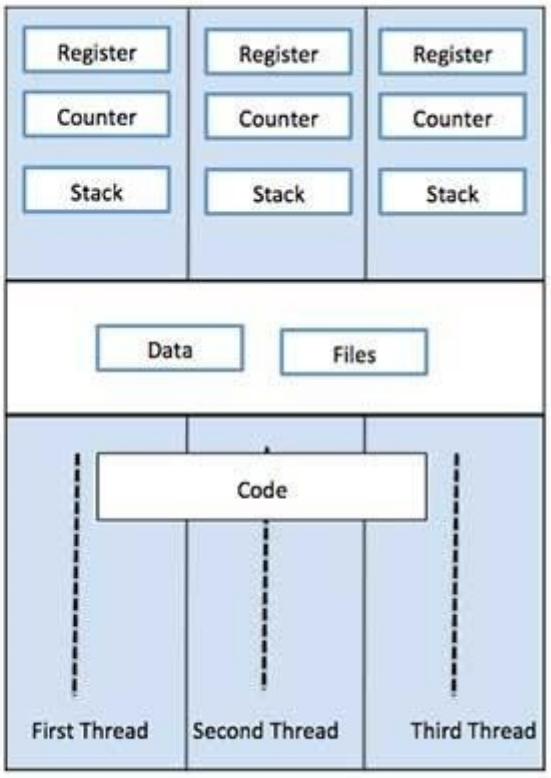
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

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.



Single Process P with single thread



Single Process P with three threads

Difference between Process and Thread

S.N.	Proces s	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 operatin

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

Advantages of Thread

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

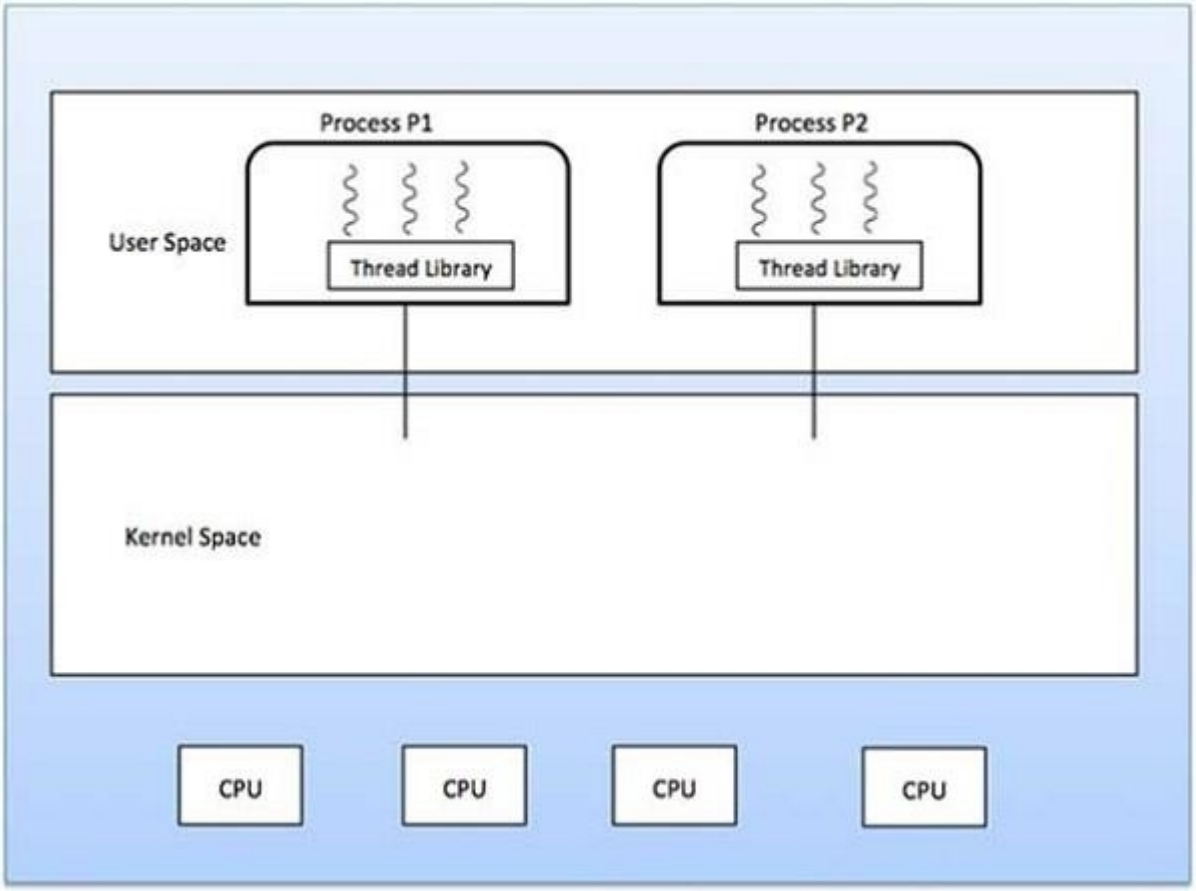
Types of Thread

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 specifiic 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

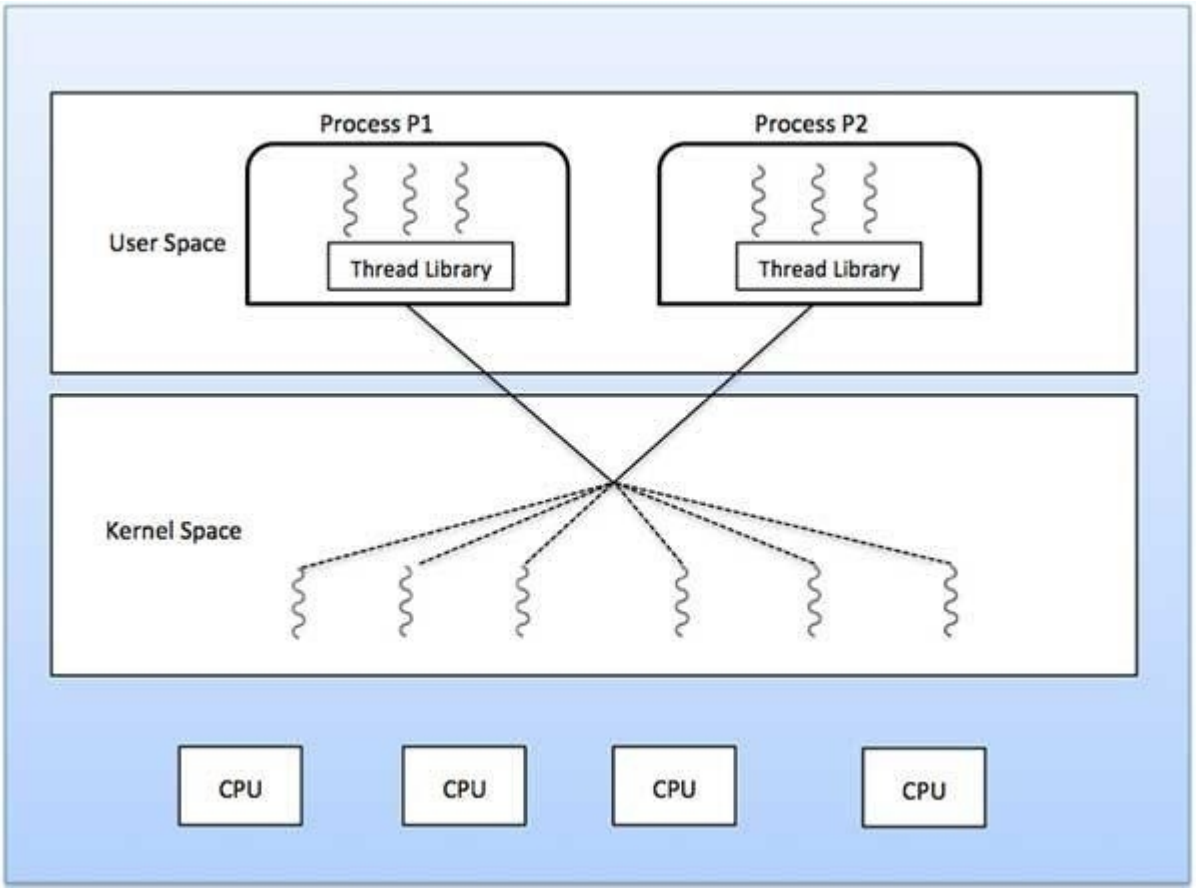
Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

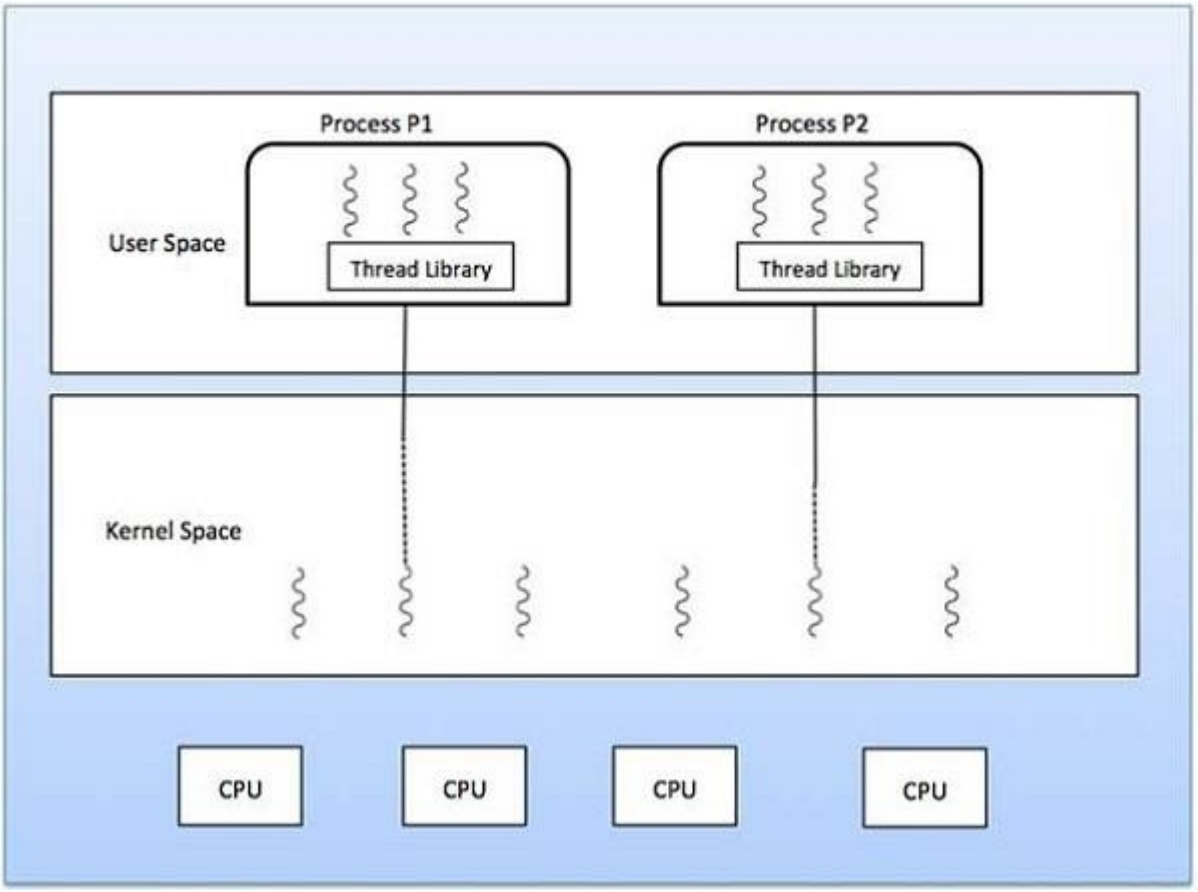
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model 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.

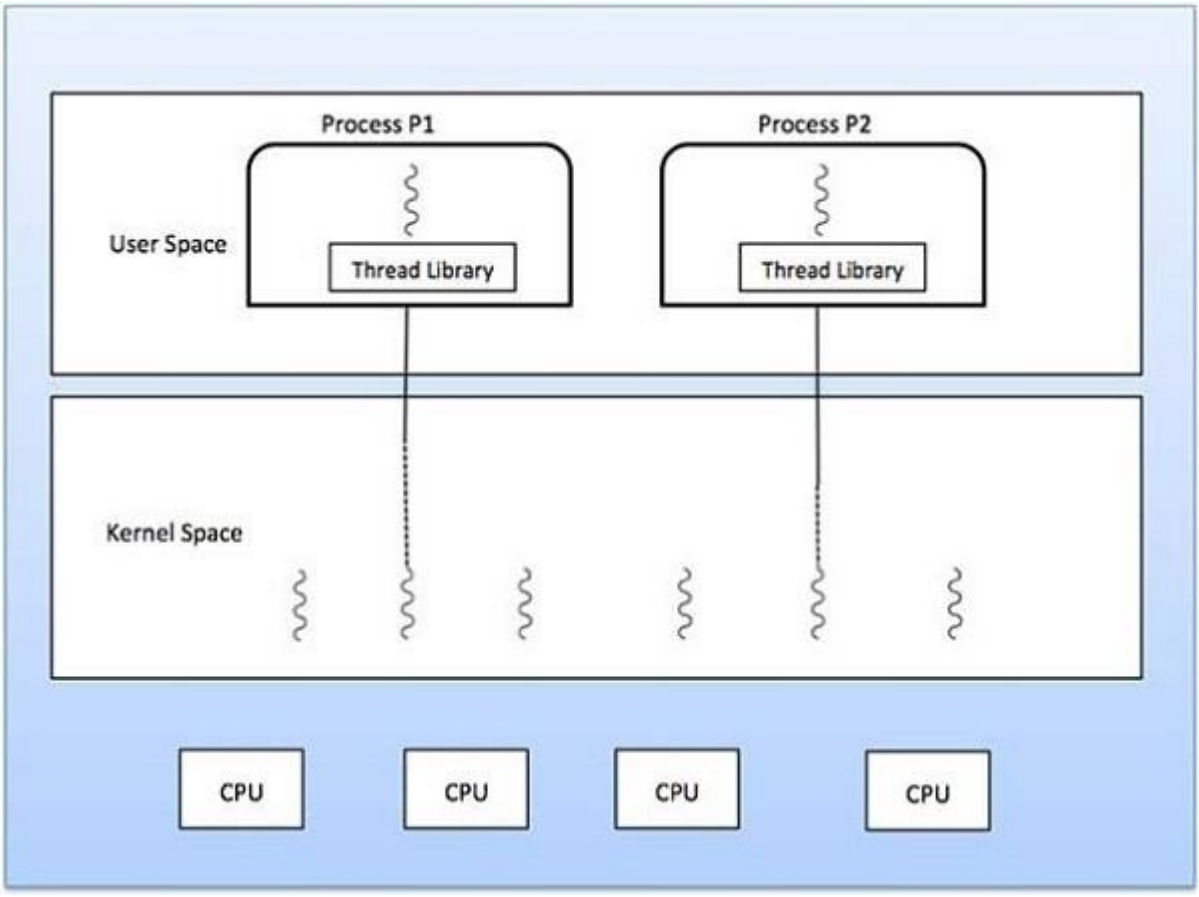
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



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

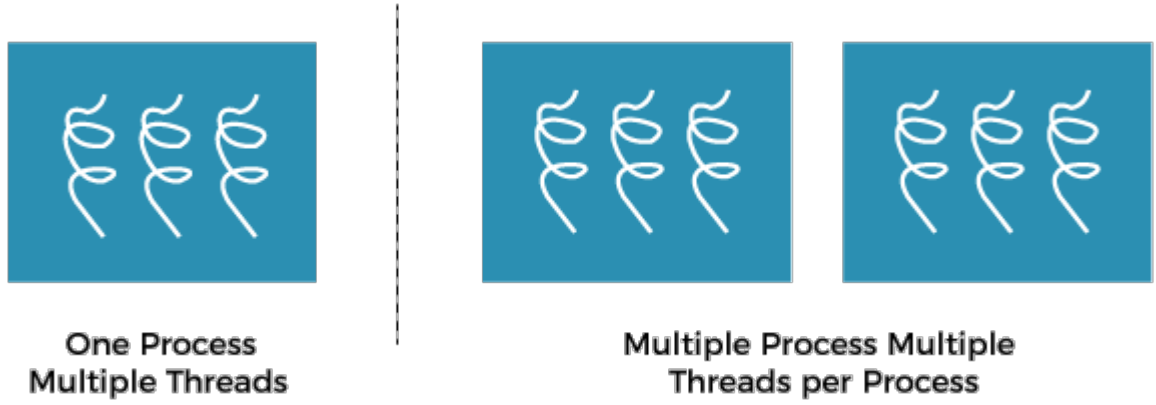
S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can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

Multithreading Models in Operating system

In this article, we will understand the multithreading model in the Operating system.

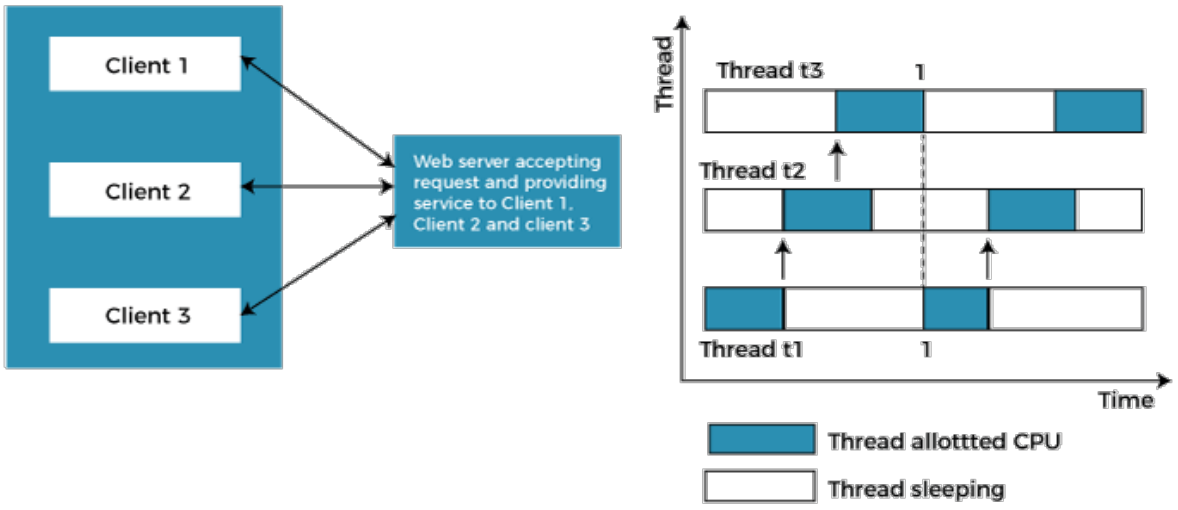
Multithreading Model:

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.

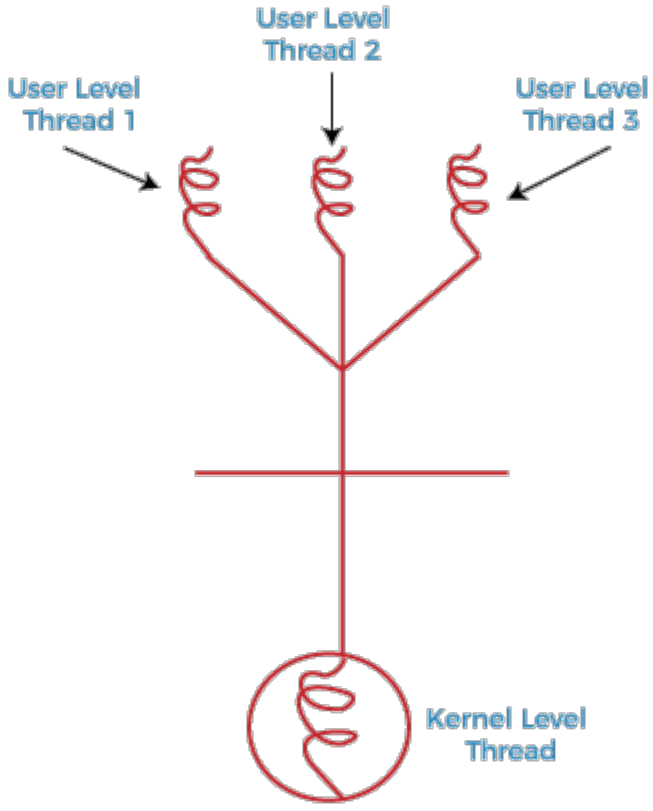
There exists three established multithreading models classifying these relationships are:

- 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 efffective 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 offfered 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.

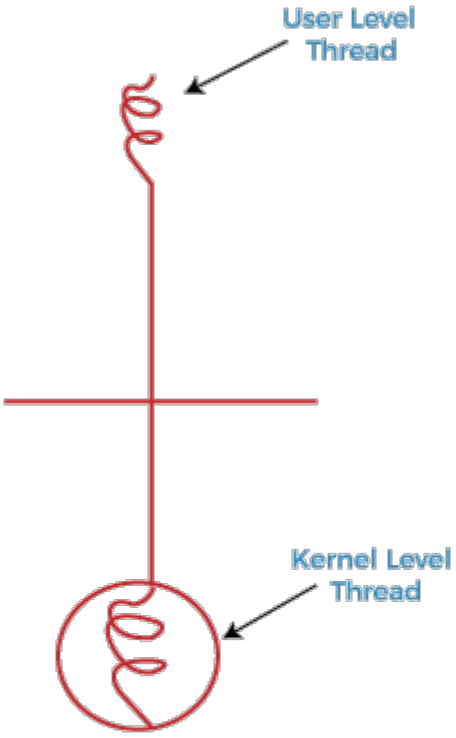


Many-to-one model

In the above fiigure, 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 benefiit 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.

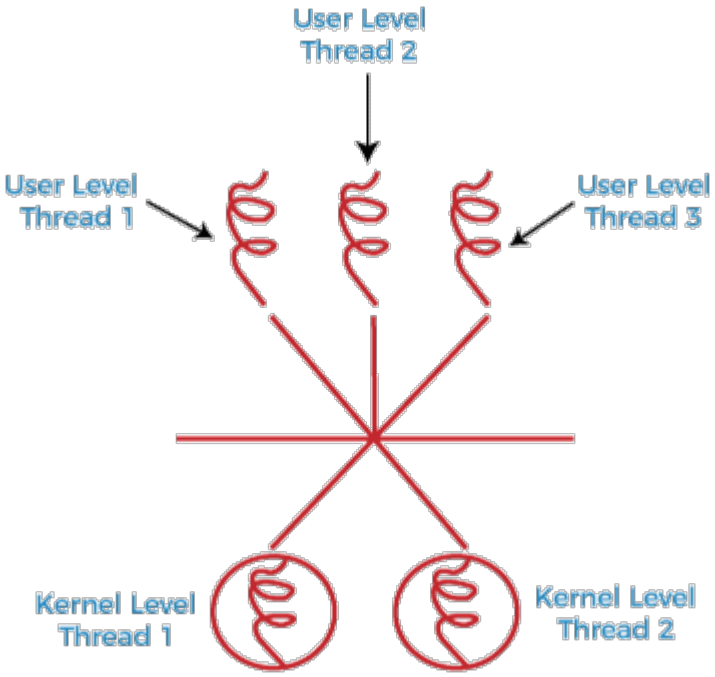


One-to-one model

In the above fiigure, 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.



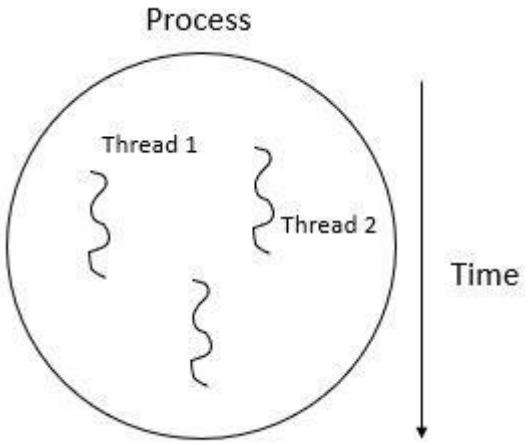
Many-to-Many model

Many to many versions of the multithreading model associate several user-level threads to the same or much less variety of kernel-level threads in the above fgure.

What are thread libraries?

A thread is a lightweight of process and is a basic unit of CPU utilization which consists of a program counter, a stack, and a set of registers.

Given below is the structure of thread in a process −



A process has a single thread of control where one program can counter and one sequence of instructions is carried out at any given time. Dividing an application or a program into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

Thread has the ability to share an address space and all of its data among themselves. This ability is essential for some specific applications.

Threads are lighter weight than processes, but they are faster to create and destroy than processes.

Thread Library

A thread library provides the programmer with an Application program interface for creating and managing thread.

Ways of implementing thread library

There are two primary ways of implementing thread library, which are as follows −

- The first approach is to provide a library entirely in user space with kernel support. All code and data structures for the library exist in a local function call in user space and not in a system call.
- The second approach is to implement a kernel level library supported directly by the operating system. In this case the code and data structures for the library exist in kernel space.

Invoking a function in the application program interface for the library typically results in a system call to the kernel.

The main thread libraries which are used are given below −

- **POSIX threads** − Pthreads, the threads extension of the POSIX standard, may be provided as either a user level or a kernel level library.
- **WIN 32 thread** − The windows thread library is a kernel level library available on windows systems.
- **JAVA thread** − The JAVA thread API allows threads to be created and managed directly as JAVA programs.