Multithreading Models

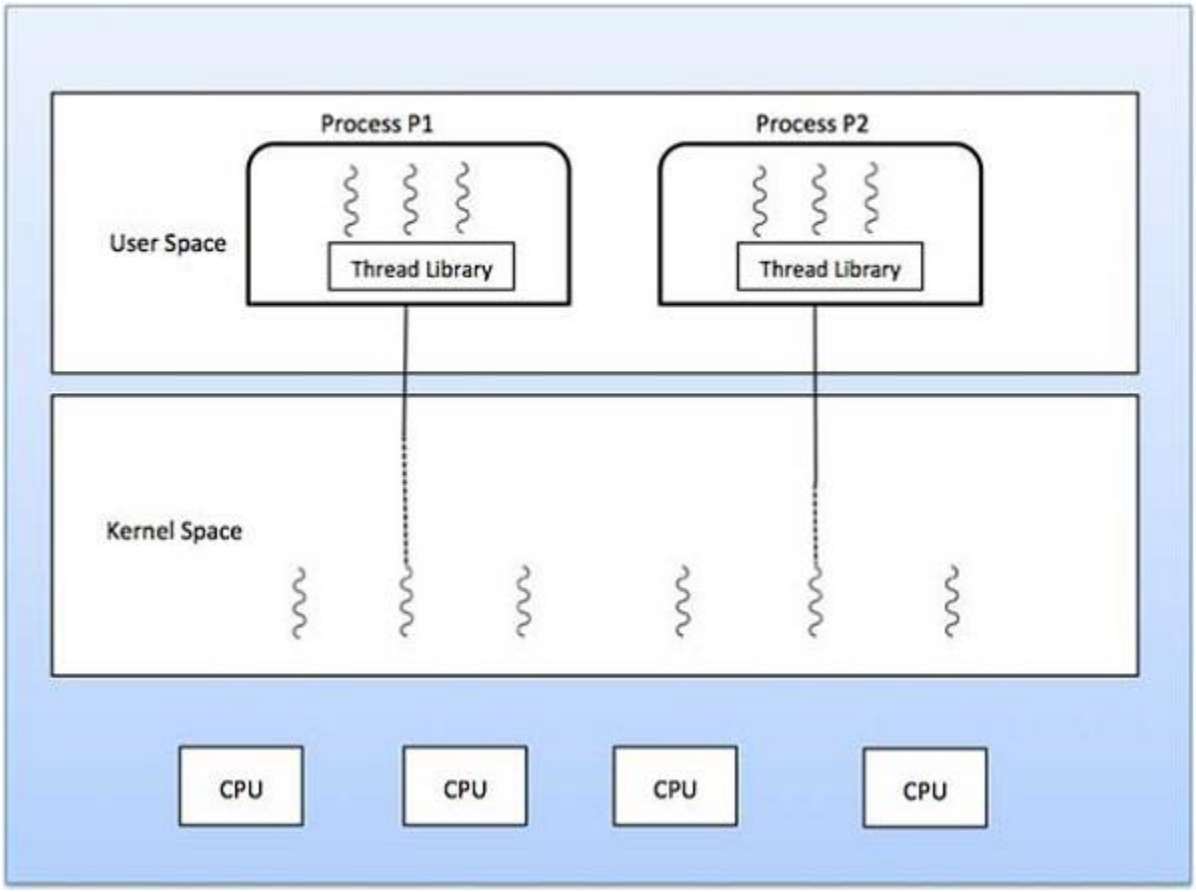
Some operating systems provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

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

Many to 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.

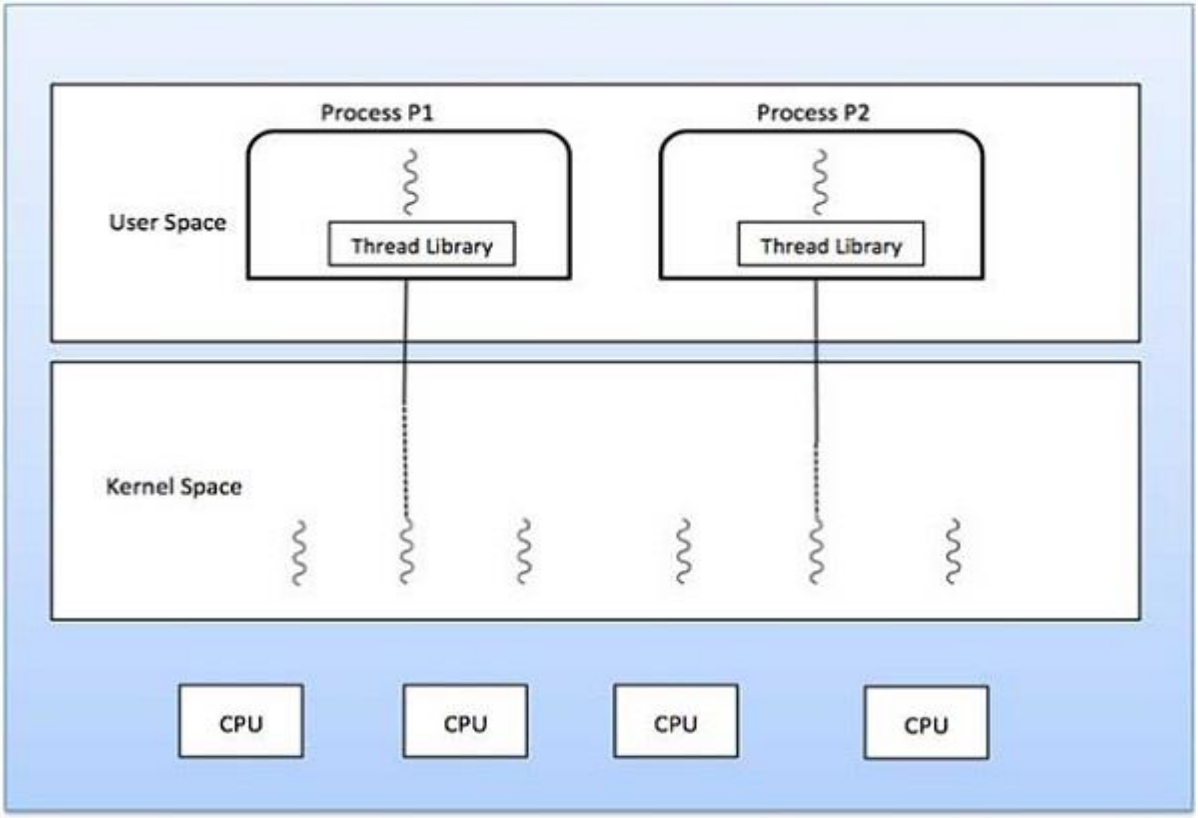
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



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.

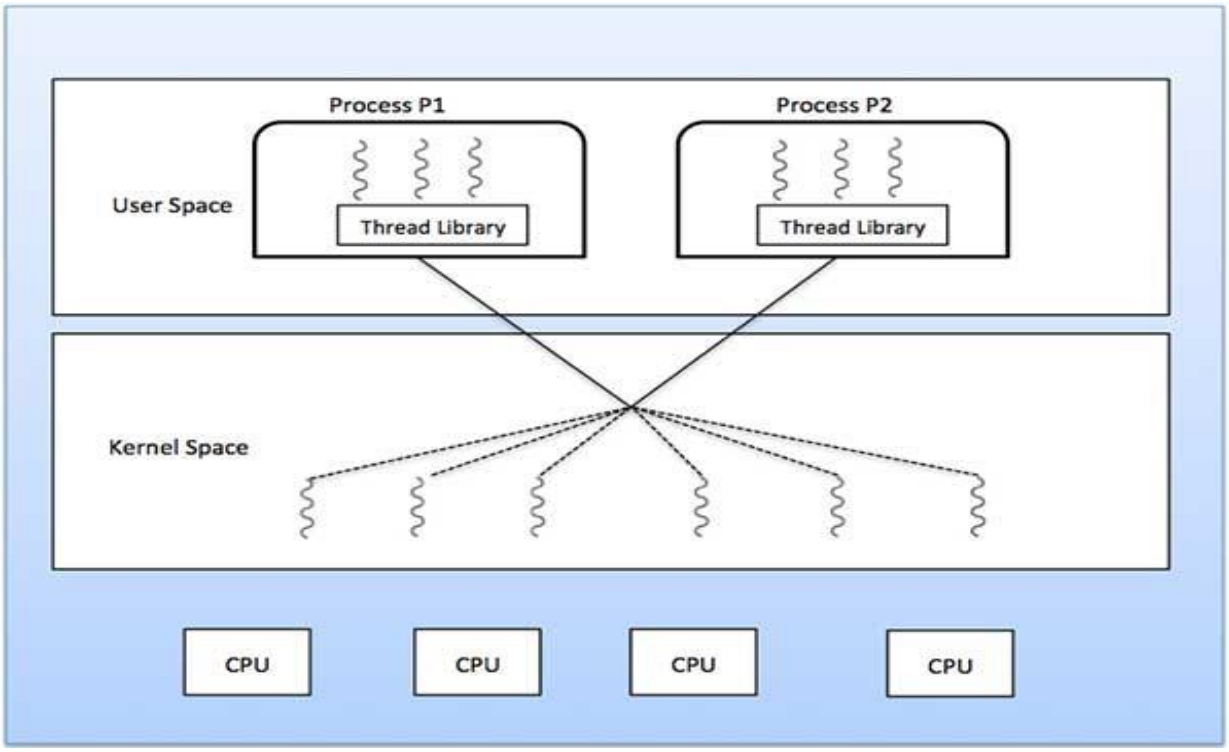


Many-to-Many Model

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 many-to-one model, a developer can create as many user threads as he wishes but true concurrency is not gained because the kernel can schedule only one thread at a time. The one-to-one model allows for greater concurrency but the developer has to be careful not to create too many threads within one application. The many-to-many model suffers from neither of these shortcomings because 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.



Thread Cancellation

Thread cancellation is the task of terminating a thread before it has completed. For example, if multiple threads are concurrently searching a database and one thread returns the result, the remaining threads might be canceled. A thread that is to be cancelled if often referred to as the target thread.

Cancellation of a target thread may occur in two different scenarios:

- **Asynchronous cancellation**: one thread immediately terminates the target thread.
- **Deferred cancellation**: The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

The difficulty with cancellation occurs when resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operation system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, cancelling a thread asynchronously may not free a necessary system-wide resource.

With deferred cancellation, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine if it should be canceled or not. This allows a thread to check whether it should be canceled at a point when it can be canceled safely. Threads refer to such points as **cancellation point**.

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.