**Report on the implementation of threading and synchronization applications**

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 that hold its current working variables, and a stack that contains the execution history.

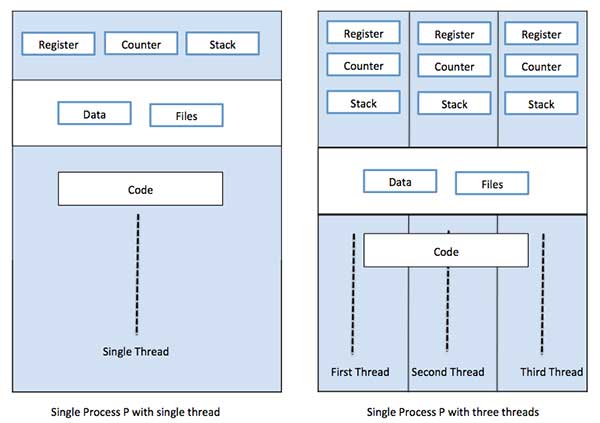
A thread shares with its peer threads a few pieces of information like code segment, data segment, and open files. When one thread alters a code segment memory item, all other threads see that.

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 the performance of the 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 the web server. They also provide a suitable foundation for the parallel execution of applications on shared memory multiprocessors. The

The following figure shows the working of a single-threaded and a multithreaded process.



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 the utilization of multiprocessor architectures to a greater scale and efficiency.

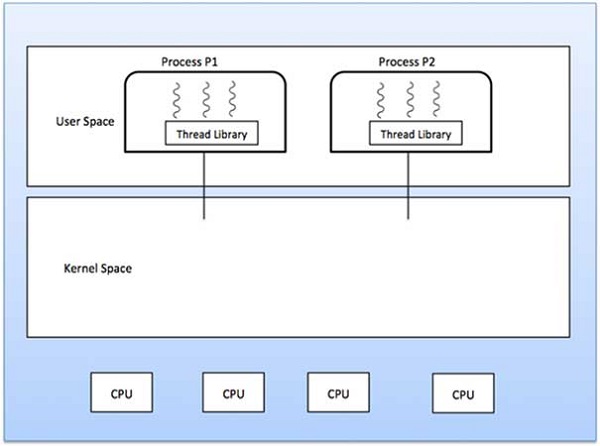
Types of Thread

Threads are implemented in the following two ways −

* **User Level Threads** – User-managed threads.
* **Kernel Level Threads** − Operating System managed threads acting on the 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, passing messages and data between threads, 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 blocked.
* Multithreaded applications 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 individual threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling, and management in the Kernel space. Kernel threads are generally slower to create and manage than 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 user threads.
* Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

**Multithreading Models:**

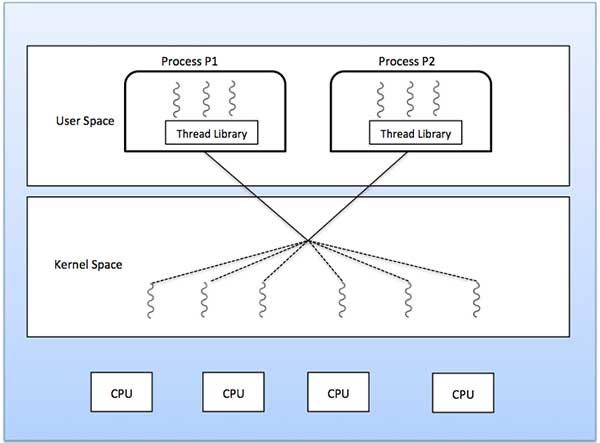
Some operating systems provide a combined user-level thread and Kernel-level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

* Many to many relationships.
* Many one relationships.
* 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.

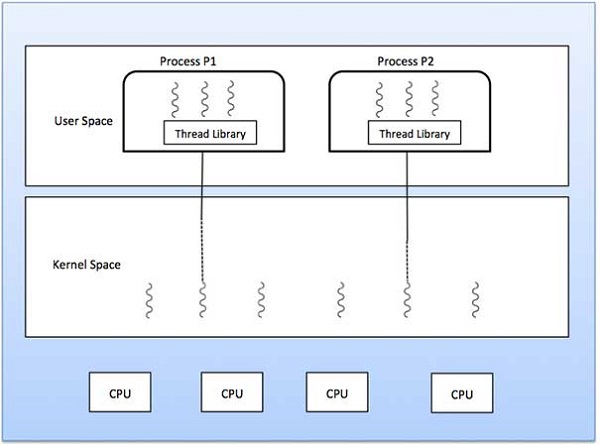
The following diagram shows the many-to-many threading model where 6 user-level threads are multiplexing with 6 kernel-level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model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

The 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 a 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.

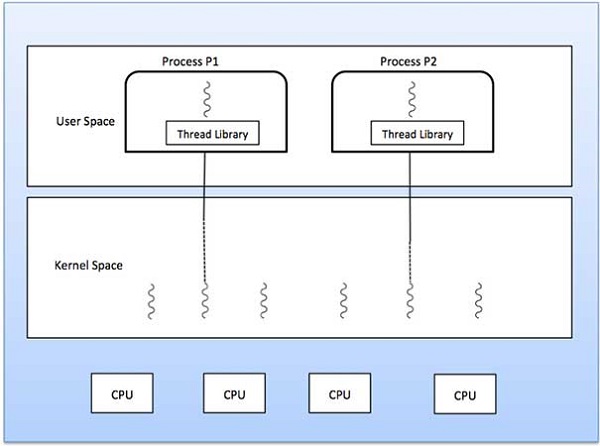
If the user-level threads libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



**One-to-One Model:**

There is a one-to-one relationship between the 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.

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

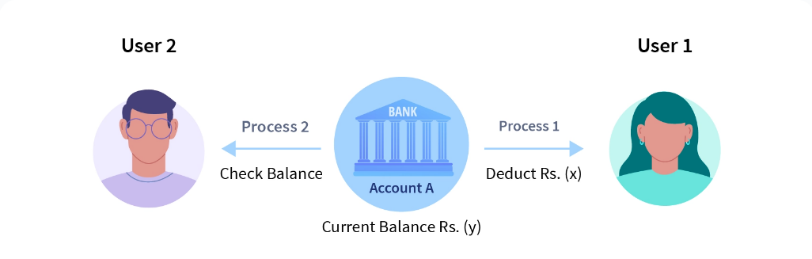


**Synchronization:**

What is Process Synchronization in OS?

An operating system is a software that manages all applications on a device and basically helps in the smooth functioning of our computer. Because of this reason, the operating system has to perform many tasks, and sometimes simultaneously. This isn't usually a problem unless these simultaneously occurring processes use a common resource.

For example, consider a bank that stores the account balance of each customer in the same database. Now suppose you initially have x rupees in your account. Now, you take out some amount of money from your bank account, and at the same time, someone tries to look at the amount of money stored in your account. As you are taking out some money from your account, after the transaction, the total balance left will be lower than x. But, the transaction takes time, and hence the person reads x as your account balance which leads to inconsistent data. If in some way, we could make sure that only one process occurs at a time, we could ensure consistent data.

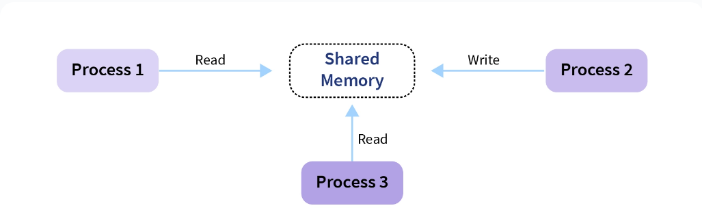


In the above image, if Process1 and Process2 happen at the same time, user 2 will get the wrong account balance as Y because of Process1 is being transacted when the balance is X.

Inconsistency of data can occur when various processes share a common resource in a system which is why there is a need for process synchronization in the operating system.

How does Process Synchronization in OS work?

Let us take a look at why exactly we need Process Synchronization. For example, If a process1 is trying to read the data present in a memory location while another process2 is trying to change the data present at the same location, there is a high chance that the data read by process1 will be incorrect.



Let us look at different elements/sections of a program:

* **Entry Section:** The entry Section decides the entry of a process.
* **Critical Section:** Critical section allows and makes sure that only one process is modifying the shared data.
* **Exit Section:** The entry of other processes in the shared data after the execution of one process is handled by the Exit section.
* **Remainder Section:** The remaining part of the code which is not categorized as above is contained in the Remainder section.

**Requirements of Synchronization:**

The following three requirements must be met by a solution to the critical section problem:

* **Mutual exclusion:** If a process is running in the critical section, no other process should be allowed to run in that section at that time.
* **Progress:** If no process is still in the critical section and other processes are waiting outside the critical section to execute, then any one of the threads must be permitted to enter the critical section. The decision of which process will enter the critical section will be taken by only those processes that are not executing in the remaining section.
* **No starvation:** Starvation means a process keeps waiting forever to access the critical section but never gets a chance. No starvation is also known as Bounded Waiting.
  + A process should not wait forever to enter inside the critical section.
  + When a process submits a request to access its critical section, there should be a limit or bound, which is the number of other processes that are allowed to access the critical section before it.
  + After this bond is reached, this process should be allowed to access the critical section.

### Mutex Locks:

Implementation of Synchronization hardware is not an easy method, which is why Mutex Locks were introduced.

Mutex is a locking mechanism used to synchronize access to a resource in the critical section. In this method, we use a LOCK over the critical section. The LOCK is set when a process enters from the entry section, and it gets unset when the process exits from the exit section.

**Conclusion:**

* Synchronization is the effort of executing processes such that no two processes have access to the same shared data.
* Four elements of program/data are:
  + Entry section
  + Critical section
  + Exit section
  + Reminder section
* The critical section is a portion of code that a single process can access at a specified moment in time.
* Three essential rules that any critical section solution must follow are as follows:
  + Mutual Exclusion
  + Progress
  + No Starvation(Bounded waiting)
* Solutions to critical section problems are:
  + Peterson's solution
  + Synchronization hardware
  + Mutex Locks
  + Semaphore