Threading

What is Thread?

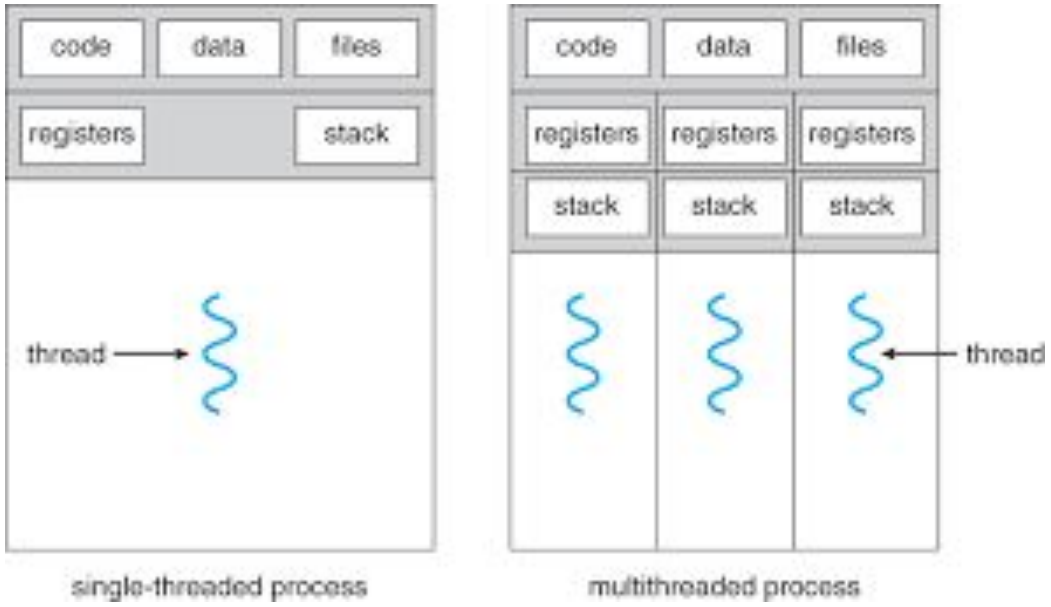
Within a program, a thread is a separate execution path. It is a lightweight process that the operating system can schedule and run concurrently with other threads. The operating system creates and manages threads, and they share the same memory and resources as the program that created them. This enables multiple threads to collaborate and work efficiently within a single program.

Each thread belongs to exactly one process.

Why Multithreading?

Multithreading is a technique used in operating systems to improve the performance and responsiveness of computer systems. Multithreading allows multiple threads (i.e., lightweight processes) to share the same resources of a single process, such as the CPU, memory, and I/O devices.

Single threaded Process vs Multi-threaded Process



Difference between Process and Thread

Parameter	Process	Thread
Definition	Process means a program is in execution.	Thread means a segment of a process.
Lightweight	The process is not Lightweight.	Threads are Lightweight.
Termination time	The process takes more time to terminate.	The thread takes less time to terminate.
Creation time	It takes more time for creation.	It takes less time for creation.
Communication	Communication between processes needs more time compared to thread.	Communication between threads requires less time compared to processes.
Context switching time	It takes more time for context switching.	It takes less time for context switching.
Resource	Process consume more resources.	Thread consume fewer resources.
Treatment by OS	Different process are tread separately by OS.	All the level peer threads are treated as a single task by OS.
Memory	The process is mostly isolated.	Threads share memory.
Sharing	It does not share data	Threads share data with each other.

Advantages of Thread

- ***Responsiveness:*** If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
- ***Faster context switch:*** Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.
- ***Effective utilization of multiprocessor system:*** If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.

- ***Resource sharing:*** Resources like code, data, and files can be shared among all threads within a process. Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.
- ***Communication:*** Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two process.
- ***Enhanced throughput of the system:*** If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

Disadvantages of threads

- Thread synchronization is an extra over head to the developers.
- Shares the common data across the threads might cause the data inconsistency or thread sync issues.
- Threads blocking for resources is more common problem.
- Difficult in managing the code in terms of debugging or writing the code.

Types of Threads

There are two types of threads:

- User Level Thread
- Kernel Level Thread

User Level Thread vs Kernel Level Thread

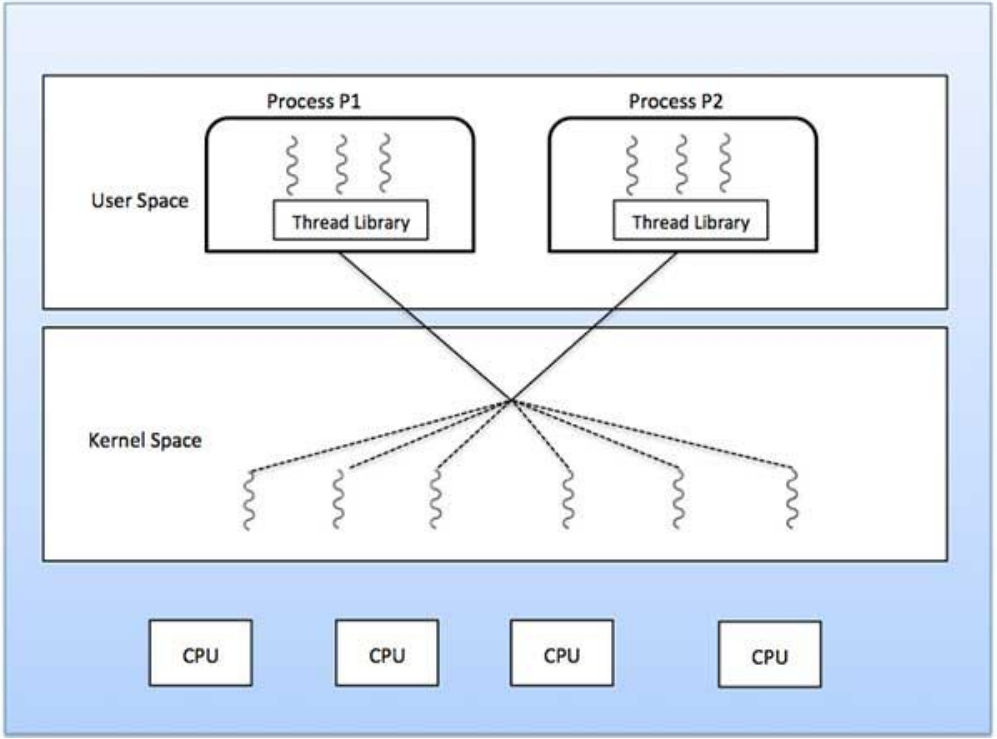
S.N.	User Level Threads	Kernel Level Thread
1	User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User level thread is generic and can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

Multithreading Models

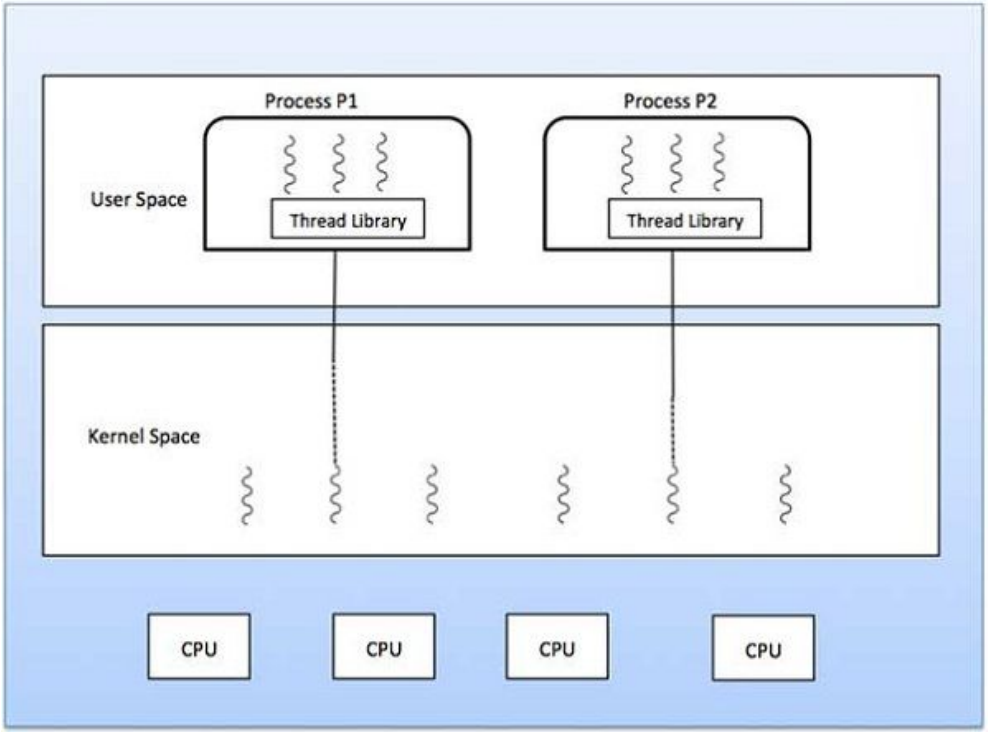
Multithreading models are three types

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

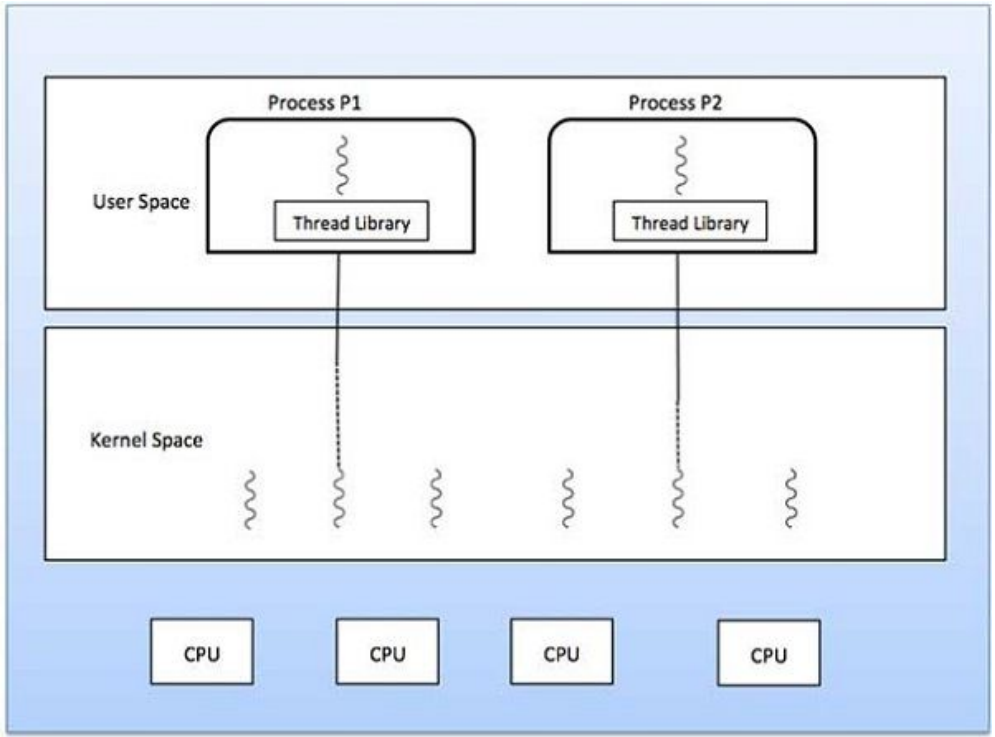
Many to Many Model



Many to One Model



One to One Model



Common Problems while using threads

1. Race condition : Unsynchronized access to shared memory can introduce race conditions.

2. Deadlocks : A situation in which two or more processes are unable to proceed because each is waiting for one the others to do something.

3. Live Locks : A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work:

Ex: A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

4. Starvation : A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Thread Synchronization

Threads share same address space, that means they can have access to the shared resources. There are times when you might want to coordinate access to shared resources. For example, in a database system, you might not want one thread to be updating a database record while another thread is trying to read it. Synchronization ensures serialized access to the shared data. To achieve this locking and unlocking of code blocks are used. The piece of code that comes in between lock and unlock is called critical section.

How to Synchronize

1. Mutual-Exclusion
2. Condition variables
3. Barriers
4. Semaphores

1. Mutexes

Mutex is short for MUTual EXclusion. Unless the word is qualified with additional terms such as shared mutex, recursive mutex or read/write mutex then it refers to a type of lockable object that can be owned by exactly one thread at a time.

Types of Mutex

Recursive Mutexes: A recursive mutex is similar to a plain mutex, but one thread may own multiple locks on it at the same time.

Read/Write Mutex: Sometimes called shared mutexes, multiple-reader/single-writer mutexes or just read/write mutexes.

Spinlocks : A spinlock is a special type of mutex that does not use OS synchronization functions when a lock operation has to wait. Instead, it just keeps trying to update the mutex data structure to take the lock in a loop.

2. Condition Variables

A condition variable is basically a container of threads that are waiting for a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.

3. Barriers

A barrier is a type of synchronization method. A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

4. Semaphores

A Semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions. The variable is then used as a condition to control access to some system resource. A semaphore is a very relaxed type of lockable object. A given semaphore has a predefined maximum count, and a current count. You take ownership of a semaphore with a wait operation, also referred to as decrementing the semaphore, or even just abstractly called P. You release ownership with a signal operation, also referred to as incrementing the semaphore, a post operation.

Creating threads

Let's look at the sample code (t1.cpp).

```
#include <iostream>
#include <thread>

void thread_function()
{
    std::cout << "thread function\n";
}

int main()
{
    std::thread t(&thread_function);    // t starts running
    std::cout << "main thread\n";
    t.join();    // main thread waits for the thread t to finish
    return 0;
}
```

This code will print out (on linux system):

```
$ g++ t1.cpp -o t1 -std=c++11 -pthread
$ ./t2
thread function
main thread
```

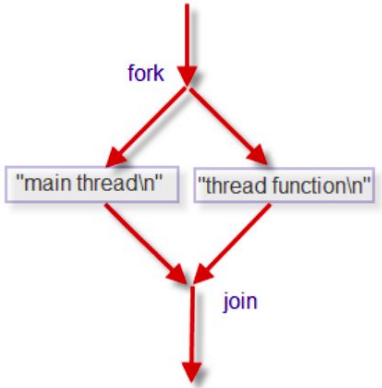
First thing we want to do is creating a thread object (worker thread) and give it a work to do in a form of a function.

The main thread wants to wait for a thread to finish successfully. So, we used join(). If the initial main thread didn't wait for the new thread to finish, it would continue to the end of main() and end the program, possibly before the new thread have had a chance to run.

While the main thread is waiting, the main thread is idling. Actually, the OS may take the CPU away from the main thread.

Note that we have a new Standard C++ Library header **#include <thread>** in which the functions and classes for threads are declared.

This is the diagram how the flow looks like.



However, in the real world, things are not that ideal and more likely to be asymmetric. Probably, it may look more like the next picture.

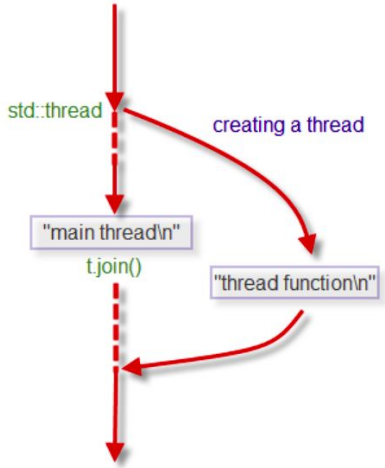
While the worker thread is starting via constructor

std::thread t, there might be overhead of creating

a thread (this overhead can be reduced by using

thread pool). The dotted line indicates a possible

blocked state.



Detaching Threads

We can make a new thread to run free to become a daemon process.

```
// t2.cpp
int main()
{
    std::thread t(&thread_function);
    std::cout << "main thread\n";
    // t.join();
    t.detach();
    return 0;
}
```

The detached child thread is now free, and runs on its own. It becomes a daemon process.

```
$ g++ t2.cpp -o t2 -std=c++11 -pthread
$ ./t2
main thread
```

Passing Parameters to a thread

```
#include <string>

void thread_function(std::string s)
{
    std::cout << "thread function ";
    std::cout << "message is = " << s << std::endl;
}

int main()
{
    std::string s = "Kathy Perry";
    std::thread t(&thread_function, s);
    std::cout << "main thread message = " << s << std::endl;
    t.join();
    return 0;
}
```

From the following output, we know the string has been passed to the thread function successfully.

```
thread function message is = Kathy Perry
main thread message = Kathy Perry
```

If we want to pass the string as a ref, we may want to try this:

```
void thread_function(std::string &s)
{
    std::cout << "thread function ";
    std::cout << "message is = " << s << std::endl;
    s = "Justin Beaver";
}
```

In fact, the message was passed by value not by reference. To pass the message by reference, we should modify the code a little bit like this using **ref**:

```
std::thread t(&thread_function, std::ref(s));
```

In fact, the message was passed by value not by reference. To pass the message by reference, we should modify the code a little bit like this using **ref**:


```
std::thread t(&thread_function, std::ref(s));
```

Thread id

We can get id information using **this_thread::get_id()**:

```
int main()
{
    std::string s = "Kathy Perry";
    std::thread t(&thread_function, std::move(s));
    std::cout << "main thread message = " << s << std::endl;

    std::cout << "main thread id = " << std::this_thread::get_id()
<< std::endl;
    std::cout << "child thread id = " << t.get_id() << std::endl;

    t.join();
    return 0;
}
```

Output:

```
thread function message is = Kathy Perry
main thread message =
main thread id = 1208
child thread id = 5224
```

The thread library provides the suggestion for the number of threads:

```
int main()
{
    std::cout << "Number of threads = "
              <<  std::thread::hardware_concurrency() << std::endl;
    return 0;
}
```

Output:

```
Number of threads = 2
```