### MULTIPROCESSING vs. MULTITHREADING

What is Multiprocessing?

In computing, a **process** is an instance of a computer program that is being executed. Or simply Running program is also called as process.

Multiprocessing means  “ Having Two or more CPU’s in a single computer system”.

For example, if a computer system has dual core, and there are two process to run(execute) concurrently(at same time),  it can be achieved by assigning each processes to each core of the system. Hence, the two process can execute simultaneously.

In other words, multiprocessing can defined as multiple process can execute at same time rather than one after other.

In Multiprocessing, each process have different address space and resources.

Different types of Multiprocessing?

SMP(Symmetric multiprocessing): All CPU’s treat equal.

**symmetric multiprocessing** (SMP) involves a multiprocessor computer hardware architecture where two or more identical processors are connected to a single shared main memory and are controlled by a single OS instance. Most common multiprocessor systems today use an SMP architecture. In the case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors. Processors may be interconnected using buses, crossbar switches or on-chip mesh networks.

**Thread-Safe:** Thread safety is a computer programming concept applicable in the context of multi-threaded programs. A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads. In particular, it must satisfy the need for multiple threads to access the same shared data, and the need for a shared piece of data to be accessed by only one thread at any given time.

Thread-safe code is code that will work even if many Threads are executing it simultaneously.

**Thread:**

A **thread** is a sequence of such instructions within a program that can be executed independently of other code.

Before we can dive in depth into threading concepts, we need to get familiarized with a few terms related to threads, parallelism and concurrency.

* **Lightweight Process** (LWP) can be thought of as a virtual CPU where the number of LWPs is usually greater than the number of CPUs in the system. Thread libraries communicate with LWPs to schedule threads. LWPs are also sometimes referred to as *kernel threads*.
* **X-to-Y** model. The mapping between LWPs and Threads. Depending upon the operating system implementation and/or user-level thread library in use, this can vary from 1:1, X:1, or X:Y. Linux, some BSD kernels, and some Windows versions use the 1:1 model. User-level threading libraries are commonly in the X:1 class as the underlying kernel does not have any knowledge of the user-level threads. The X:Y model is used in Windows 7.
* **Contention Scope** is how threads compete for system resources (i.e. scheduling).
* **Bound threads** have system-wide contention scope, in other words, these threads contend with other processes on the entire system.
* **Unbound threads** have process contention scope.
* **Thread-safe** means that the program protects shared data, possibly through the use of mutual exclusion.
* **Reentrant** code means that a program can have more than one thread executing concurrently.
* **Async-safe** means that a function is reentrant while handling a signal (i.e. can be called from a signal handler).
* **Concurrency vs. Parallelism** - They are not the same! Parallelism implies simultaneous running of code (which is not possible, in the strict sense, on uniprocessor machines) while concurrency implies that many tasks can run in any order and possibly in parallel.

### **Thread Pool (Boss/Worker)**

One thread dispatches other threads to do useful work which are usually part of a *worker thread pool*. This thread pool is usually pre-allocated before the boss (or master) begins dispatching threads to work. Although threads are lightweight, they still incur overhead when they are created.

## Protecting Shared Resources

Threads may operate on disparate data, but often threads may have to touch the same data. It is unsafe to allow concurrent access to such data or resources without some *mechanism that defines a protocol for safe access*! Threads must be explicitly instructed to block when other threads may be potentially accessing the same resources.

### Mutual Exclusion

Mutual exclusion is the method of *serializing access* to shared resources. You do not want a thread to be modifying a variable that is already in the process of being modified by another thread! Another scenario is a dirty read where the value is in the process of being updated and another thread reads an old value.   
  
Mutual exclusion allows the programmer to create a defined protocol for serializing access to shared data or resources. Logically, a **mutex** is a lock that one can virtually attach to some resource. If a thread wishes to modify or read a value from a shared resource, the thread must first gain the lock. Once it has the lock it may do what it wants with the shared resource without concerns of other threads accessing the shared resource because other threads will have to wait. Once the thread finishes using the shared resource, it unlocks the mutex, which allows other threads to access the resource. This is a protocol that serializes access to the shared resource. Note that such a protocol must be enforced for the data or resource a mutex is protecting across all threads that may touch the resource being protected. If the protocol is violated (e.g., a thread modifies a shared resource without first requesting a mutex lock), then the protocol defined by the programmer has failed. There is nothing preventing a thread programmer, whether unintentionally (most often the case, i.e., a bug -- see race conditions below) or intentionally from implementing a flawed serialization protocol.   
  
As an analogy, you can think of a mutex as a safe with only one key (for a standard mutex case), and the resource it is protecting lies within the safe. Only one person can have the key to the chest at any time, therefore, is the only person allowed to look or modify the contents of the chest at the time it holds the key.   
  
The code between the lock and unlock calls to the mutex, is referred to as a **critical section**. Minimizing time spent in the critical section allows for greater concurrency because it potentially reduces the amount of time other threads must wait to gain the lock. Therefore, it is important for a thread programmer to minimize critical sections if possible.

What is a Thread?   
A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*.   
Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.  
Threads operate faster than processes due to following reasons:  
1) Thread creation is much faster.  
2) Context switching between threads is much faster.  
3) Threads can be terminated easily  
4) Communication between threads is faster.

What are the differences between process and thread?  
Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section and OS resources like open files and signals. But, like process, a thread has its own program counter (PC), a register set, and a stack space.

Why Multithreading?

The **process** of executing multiple threads simultaneously is known as multithreading.  
Let's summarize the discussion in points: 1. The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time.

The concept of **multi-threading** needs proper understanding of these two terms – **a process and a thread**. A process is a program being executed. A process can be further divided into independent units known as threads.

A thread is like a small light-weight process within a process. Or we can say a collection of threads is what is known as a process.

