CS 240 Programming in C

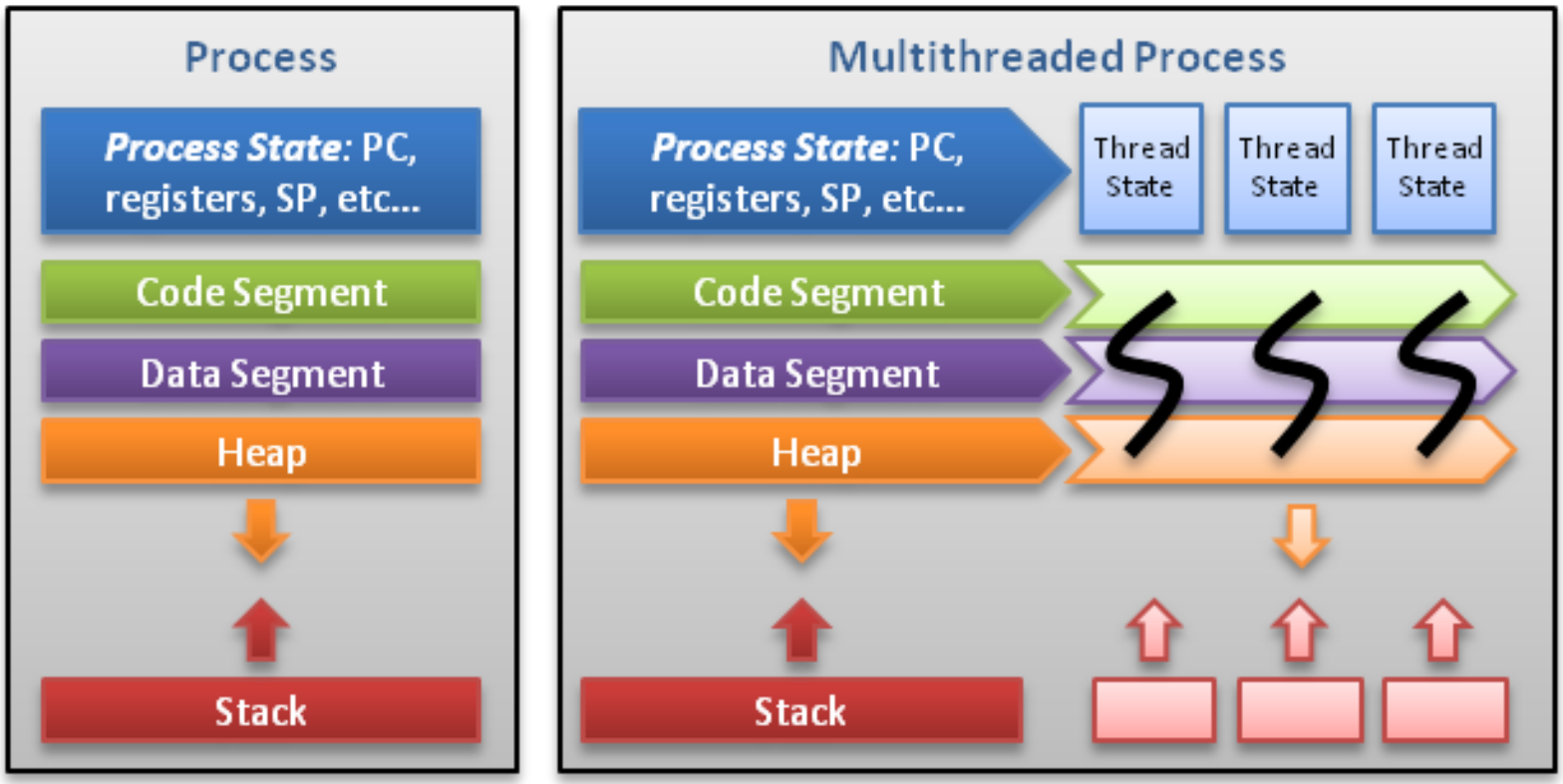
Process Control, Threads

April 20, 2023

Thread

- Thread is an execution unit that is part of a process. A process can have multiple threads, all executing at the same time. It is a unit of execution in concurrent programming. A thread is lightweight and can be managed independently by a scheduler. It helps you to improve the application performance using parallelism.

Thread



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

© Alfred Park, http://randu.org/tutorials/threads

Properties of Thread

- Single system call can create more than one thread
- Threads share data and information.
- Threads shares instruction, global, and heap regions. However, it has its register and stack.
- Thread management consumes very few, or no system calls because of communication between threads that can be achieved using shared memory.

Key Difference Between Process and Thread

- Process means a program is in execution, whereas thread means a segment of a process.
- A Process is not Lightweight, whereas Threads are Lightweight.
- A Process takes more time to terminate, and the thread takes less time to terminate.
- Process takes more time for creation, whereas Thread takes less time for creation.
- Process likely takes more time for context switching whereas as Threads takes less time for context switching.
- A Process is mostly isolated, whereas Threads share memory.
- Process does not share data, and Threads share data with each other.

Terminology

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 relationship between LWPs and Threads. This can vary from 1:1 to X:1 or X:Y depending on the operating system implementation and/or user-level thread library in use. The 1:1 model is used by Linux, some BSD kernels, and some Windows versions.
- **Contention Scope** is how threads compete for system resources (i.e. scheduling).

Terminology

- **Bound threads** have system-wide contention scope, which means they compete with other processes across the entire system. Unbound threads have process contention scope.
- **Thread-safe** means that the program protects shared data, possibly through the use of the mutual exclusion.
- **Reentrant code** means that a program can have more than one thread executing concurrently.
- **Async-safe** means that a function is asynchronous-safe, or asynchronous-signal-safe if it can be called safely and without side effects from within a signal handler context.
- **Concurrency vs. Parallelism** - They are not the same! Parallelism implies the 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 Synchronization

Thread synchronization is the concurrent execution of two or more threads that share critical resources. Threads should be synchronized to avoid critical resource use conflicts. Otherwise, conflicts may arise when parallel-running threads attempt to modify a common variable at the same time.
The threads library provides three synchronization mechanisms:

- mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
- joins - Make a thread wait till others are complete (terminated).
- condition variables - data type pthread_cond_t

Mutual Exclusion

Mutual exclusion is a method of serializing access to shared resources. You don't want one thread modifying a variable that is already being modified by another thread! Another scenario is a dirty read, in which the value is being updated and another thread reads an old value.

Mutual Exclusion

The mutual exclusion allows the programmer to create a defined protocol for serializing access to shared data or resources. 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 the thread finishes using the resource, it unlocks the mutex, which allows other threads to access the resource. Such a protocol must be enforced across all threads that may touch the resource being protected. Logically, a mutex is a lock that one can virtually attach to some resource.

Mutual Exclusion

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.

Demo

Sample Programs

Resource Links

- https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_26.html
- https://www.guru99.com/difference-between-process-and-thread.html
- https://www.geeksforgeeks.org/multithreading-c-2/
- https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
- http://www.csc.villanova.edu/ mdamian/threads/posixthreads.html
- https://randu.org/tutorials/threads/