Operating Systems
COMS W4118
Lecture 9

Alexander Roth

2015 – 02 – 19

1 POSIX threads

1.1 Create, Exit, Join

- `pthread_t` is an implementation type that is system-dependent.
- Linux ignores the notion of threads. Threads are just processes that share memory spaces with the parent process.
- In modern Linux, threads are now a different concept from processes.
- The `ps` command evolved from the old days of UNIX.
- `LWP` - Light Weight Process ID is the same thing as thread ID.
- The leader thread shares the same process ID and the same thread ID.
- Linux uses something like process ID for thread ID. It appears they come from the same pool of numbers.
- All threads from within a process share a process.

1.2 Mutex

- An object you declare and initialize
- Behaves like a lock.
- You declare a variable of the `mutex` type and you pass in the attributes to it.
- There is a timeout version of the mutex api, where if during the time the mutex is unable to lock, then it will throw an error.
- `trylock` is a non-blocking version of `block`.

- Semaphore has a number and does not belong to any processes.
- Mutexes are primarily locking mechanisms.
- Semaphores have more ability to share locks across threads, you cannot do that through mutexes.

1.3 Deadlock

- Deadlock is a condition where you try to lock a mutex, but for some reason the mutex isn't able to lock.
- One thread tries to lock the same mutex twice.
- You have two threads and two mutexes A and B. When thread 1 locks mutex A successfully, it tries to lock mutex B. However, right before mutex B gets locked, thread 2 locks mutex B and tries to lock mutex A.
- A way to avoid this is to make sure all threads lock mutexes in the same order.
- Order must be preserved otherwise there will be a race condition.

1.4 Reader-Writer Lock

- Very useful because a large number of applications fall into this character.
- If you have some piece of data, such as database code that is multithreaded.
- We have a file that represents the data.
- If we update a record, we need to write to this file.
- If we read from the file, we do not need to write to this file. We are just accessing it.
- Everyone can read at the same time because nothing it getting updated.
- If we use a mutex, we will have a build up in time taken to read the file if everyone is just reading.
- Using a `rwlock` allows you to lock the file in read or write mode.
- Write lock is exclusive. You lock in write mode and no other processes can proceed.
- If you grab the lock in read mode, it's ok for other threads to call read lock and lock it at the same time.
- If everyone is locking in `rdlock`, then there is no locking for processes that are just reading.

- If one thread grabs a read lock and before it unlocks the process, a write lock is called. The write lock will not be allowed in as the writer must wait until all reading processes are finished.
- If there is a continuous stream of read locks, then no write locks will not be able to enter into the file.
- One method to avoid this problem is when the writer is called, block all new read locks.
- While a writer is pending, the subsequent readers would have to wait until the writer lock is done.
- Incrementing and decrementing a number is a non-atomic method, so we need to surround it with a lock.

1.5 Condition Variables

- You have this variable that the thread can wait on.
- The variable status changes when the thread calls signal or broadcast.
- Suppose you have a queue of data items, this queue is in shared memory.
- When a thread makes a condition happen, it will notify all other threads that this condition has occurred.