CS 471 Operating Systems

Yue Cheng

George Mason University

Fall 2019

Midterm I

- Thursday, Sep 26, 1:30pm – 2:45pm
  - 75 min, closed book, closed note

- Covering topics from lec-0 to lec-2c
  - Process vs. thread
  - `fork()`, `pthread_create()`
  - Race condition, spin lock, semaphore, CV
  - Deadlock and Starvation

Process Creation in Linux

- System call `fork()`
  - The return value of `fork()`

- Process tree

- See example problems in HW1

Process vs. Thread

- Multiple threads within a process share
  - The memory address space
  - Open files
  - Global variables, etc.

- Why thread abstraction?
  - Efficient utilization of the multi-/many-core architecture with only one process (Moore's law ending)
  - Efficient resource sharing, fast and flexible inter-thread communication
  - Less context switching overheads

Pthread

- Creating child threads using `pthread_create()`

- Parent thread waits for a certain child thread to terminate on `pthread_join()`

- Spawning multiple child threads, the execution order of each child thread is non-deterministic

Race Conditions

- Multiple processes or threads are writing to and reading from some shared data, and final result depends on who runs precisely when
  - This situation is called a race condition

- To protect shared data and guarantee mutual exclusion
  - We can use spin locks
  - We can use semaphores
  - We can use condition variables

Spin Locks

- A simple implementation of a spin lock
  - Provide mutual exclusion with atomic instruction `TestAndSet()`
  - Busy waiting: the waiting process/thread loops (spins) continuously at the entry point, until the lock is released
- Disadvantages?
  - Fairness?
  - Performance?
- Use binary locks to protect shared data structures

Semaphores

- Motivation: avoid busy waiting by blocking a process until some condition is satisfied

- Two operations
  - `sem_wait(s)`: decrease the value of s by 1, the caller is blocked with value < 0
  - `sem_post(s)`: increase the value of s by 1, if one or more process/thread is waiting, wake one

Condition Variables

- CV: an explicit queue that threads can put themselves when some condition is not as desired (by waiting on that condition)

- `cond_wait(cond_t *cv, mutex_t *lock)`
  - assume the lock is held when `cond_wait()` is called
  - puts caller to sleep + **release** the lock (atomically)
  - when awaken, **reacquires** lock before returning

- `cond_signal(cond_t *cv)`
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, just return, **doing nothing**

Condition Variables (cont.)

- Traps when using CV
  - A `cond_signal()` may only wake one thread, though multiple are waiting
  - Signal on a CV with no thread waiting results in a lost signal

- Good rules of thumb when using CV
  - Always do wait and signal while holding the lock
  - Lock is used to provide mutual exclusive access to the shared variable
  - `while()` is used to always guarantee to re-check if the condition is being updated by other thread

Deadlock and Starvation

- Subtle difference between deadlock and starvation
  - Once a set of processes are in a deadlock, there is **no future execution sequence** that can get them out of it!
  - In starvation, there does exist **hope** – some execution order may be favorable to the starving process although no guarantee it would ever occur
  - Rollback and retry are prone to starvation
  - Continuous arrival of higher priority process is another common starvation situation

Classic Problems of Synchronization

- Producer-consumer problem (CV-based version)
- Readers-writers problem
- Five dining philosophers problem
- **Goal is to gain a deep understanding of how to use CVs and semaphores, through examples**