Operating Systems
COMS W4118
Lecture 13

Alexander Roth

2015 – 03 – 05

1 Thread Design Issues

- The underlying kernel will choose any random thread to run a program.
- The thread is chosen from any underlying thread that is willing to accept a signal.

2 Synchronization

- `++` and `--` are not thread-safe operators.
- `taskset` is a program you can use that runs other programs while setting the specific CPU to use.

2.1 Race Conditions

- Hard to reproduce unless you are using multiple CPU's.
- Non-deterministic.
- We can attempt to fix the scheduling of the operating system, so that the same thing happens every time.
- If you can reproduce the exact scheduling from time 0 to some point, there is a good chance you can reproduce the error.

2.2 How to Avoid Race Conditions?

- We can use locking.
- Atomic operations (all or nothing) would prevent operations from being interleaved.

- Create a super instruction that does what we want atomically.
- We cannot have an atomic instruction for every single possible case.
- **Critical sections** are sections of code that cannot be broken up. They must be atomic.
- **Liveness** cannot allow a deadlock to occur.
- We strive for **Safety**, **Liveness**, and **Bounded waiting** from our threading environment.
- It is possible for an operating system to not schedule one thread for a while.
- An operating system may not schedule threads in a fair, distributed fashion.
- Thus, the synchronization method must work for any scenario, even if the thread is running slowly.
- Critical sections should be **efficient**, **fair**, and **simple**.
- Busy waiting (or spin locking) happens often in the kernel. Look it up.
- Efficiency must be considered in context. There is not one absolute case.
- Fairness implies that a thread should not wait too long to run.
- Complex API's are always doomed for failure, so be sure to avoid this issue.

2.3 Implementing Critical Sections

2.3.1 Version 1: Disable Interrupts

- Concurrency issues happen when signals interrupt threads.
- Timer interrupts cause processes to halt and jump between processes.
- Disabling interrupts only makes sense in the kernel, not the user space.
- If you disable signal interrupt on a single CPU, the running program will run forever.
- Disabling interrupts allows a set of operations to become atomic.
- This style of coding does not work on multiprocessors.
- Disabling interrupts ruins efficiency on the system, so it's generally a bad idea.
- This is legitimate style on uni-processing systems, typically when the kernel needed to lock something and do something.

2.3.2 Version 2: Software Locks

- There are a number of software algorithms that we can implement to provide a locking mechanism.
- The **Peterson's algorithm** is a software-based lock algorithm that does not require anything special from the hardware.
- The only assumptions needed are:
  1. that loads and stores must be atomic.
  2. We assume hardware operations execute in a serial order.
  3. We do not require special hardware instructions.
- In general adversarial scheduler model useful to think about concurrency problems.