CS318 Project #3

Preemptive Kernel

Continuing from Project 2

- Project 2 involved:
  - Context Switch
  - Stack Manipulation
  - Saving State
- Moving between threads, but in a single threaded environment

Multi-Threading

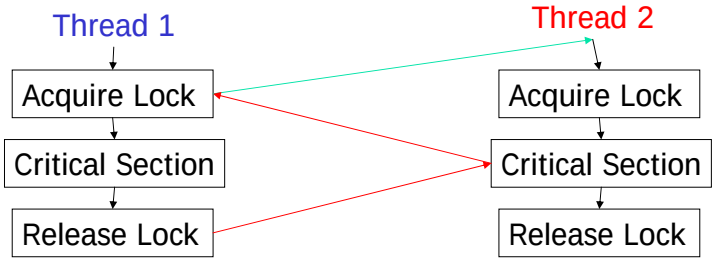
- Now you have to deal with the possibility of your context switch being interrupted
  - Timer Interrupt (10 ms)
  - Page Fault
  - TLB Miss

What do you have to do?

- Implement locking primitives (thread.c)
  - Lock
    - Acquire & Release
  - Condition
    - Signal, Wait, Broadcast
  - Semaphore
    - Up & Down
  - Barrier (Extra Credit)
    - Wait
  - Timer Interrupt (entry.S)
    - Also handle irq7 - call fake_irq7 instead of scheduler_entry

Lock

- Acquiring and releasing a lock must be done atomically
- Otherwise ...



Lock (cont.)

- Acquiring and Releasing of the lock should be done in a single threaded context
  - How do you do this?
  - Turn interrupts off for as small a window as possible

Condition Variables

- Wait operation cause the current thread to block on put itself of a wait list
- Signal causes a random thread on the wait list to start running
- Broadcast wakes up all waiting threads
- Condition Variables can “lose” signals if no thread is on the wait list to receive it

Semaphore

- Use when you need a way to store wakeup signals
  - Producer-Consumer Problem
- Two operations
  - Up -> Called to wakeup a thread, if no thread is ready to be woken up, the wakeup signal is stored
  - Down -> Called to see if a wakeup signal is stored, otherwise sleep

Mutex

- A semaphore used for mutual exclusion
  - Semaphore should only ever have values 0 and 1
- Useful when only one thread should execute a piece of code at a time

Barrier

- Used to synchronize multiple threads at a single point
  - Usually the boundary between passes of phases of an algorithm
  - Count number of threads stopped
  - Count number of threads started

Scheduling

- With preemptive kernel we can implement scheduling algorithms
  - Round-Robin (default)
  - Priority (Extra Credit)
  - Priority w/Queue
  - Random

Handling Interrupts

- Handle the timer interrupt (irq0) & irq7 plus system call interrupts (traps)
  - For irq7 call fake_irq7
  - For all interrupts
    - Save the current state
    - Interrupt controller will suspend further interrupts when generating a non-system interrupt
    - Tell the Interrupt Controller to allow more interrupts
    - Process the interrupt
  - Call scheduler to move to next process on timer interrupt

The End

- Questions?