Project 3: Preemptive Scheduler

COS 318

Fall 2013

Project 3 Schedule

- Design Review
  - Monday, Oct 21
  - 10-min time slots from 10am to 7:00pm
- Due date: Sun Nov 3, 11:55pm

General Suggestions

- Project is divided into 3 phases:
  - Timer interrupt/preemptive scheduling
  - Blocking sleep
  - Synchronization primitives
- Get each phase working <u>before</u> starting on the next
- Use provided test programs to test each component
- Start as early as you can, and get as much done as possible by the design review

Project 3 Overview

- Implement preemptive scheduling:
  - Respond to timer interrupt: ***entry.S***
  - Blocking sleep: ***scheduler.c***
- Implement synchronization primitives: ***sync.c, sync.h***
  - What are the properties of condition variables, semaphores, and barriers?
  - How do you implement them race condition-free?
- Care: turn interrupts on/off properly
  - Safety and liveness properties

Test Programs

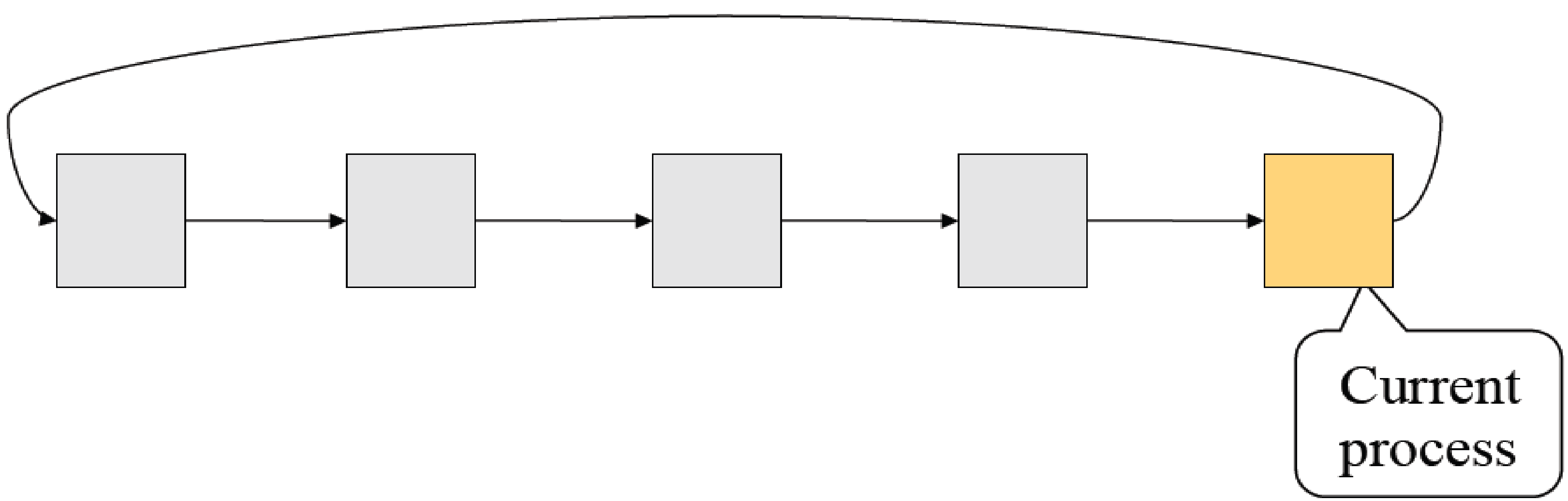
- 5 test programs provided for your convenience
- Preemptive scheduling:
  - test_regs and test_preempt
- Blocking sleep:
  - test_blocksleep
- Synchronization primitives:
  - test_barrier, test_all (tests everything, really)
- Feel free to create your own test programs!

Preemptive Scheduling

- Round-robin fashion
- Tasks are preempted via timer interrupt IRQ0
- Have time slice to determine when to preempt (time_elapsed variable in ***scheduler.c***)
- IRQ0 increments the time slice in each call

Preemptive Scheduling

- What is the workflow of one preemption cycle?
  - Have one task running, others in queue waiting
  - Save the current task before preempting
  - Change the current running task to the next one in the queue



Blocking Sleep

- Enables preemptive scheduling
- Maintain a wait queue for sleeping tasks
- When do you need to wake up the task?
  - Each task has a deadline
  - Can use time_elapsed to do the timing
  - Wake-up should happen as soon as possible
- Must handle the case when all tasks are sleeping

Synchronization Primitives

- Implement condition variables, semaphores, barriers
- What are the properties of each primitive?
  - Data structure
  - Behavior
- Ensure that you are not introducing race conditions

Review: Condition Variables

- Properties:
  - Queue of threads that are waiting on condition to become true
  - Part of a monitor (locks are implemented for you)
- Two main operations:
  - Wait: Block on a condition, release the mutex while waiting
  - Signal: Unblock since condition is true
- Broadcast operation notifies all waiting threads
- Refer to pp.13, 23 of 10/3 lecture

Review: Semaphores

- Properties:
  - Control access to a common resource
  - Value keeps track of the number of units of a resource that are currently available
  - Queue of processes that are waiting
- Two main operations:
  - Down: Decrement value, block the process
  - Up: Increment value, unblock waiting process
- Refer to p. 7-8 of 10/3 lecture

Review: Barriers

- Properties:
  - Location in code at which any thread/proc must stop until all other threads/procs reach this point
  - Keep track of number of threads at barrier, and number of threads running
  - Maintain queue of processes that are waiting
- Main operation:
  - Wait: If there are still running procs/threads, block the proc/thread. Otherwise, unblock all.
- Refer to pp. 26-28 of 10/3 lecture

Warm-Up Exercise

- Analyze implementations of synchronization primitive operations.
- Are these implementations safe?
  - Do they prevent race conditions in the kernel?
- Do these implementations preserve liveness?
  - Are the interrupts on most of the time?
- Race condition: arises when the order of execution of an operation by several different processes/threads results in unexpected behavior.