Precept 3: Preemptive Scheduler

COS 318: Fall 2020

Project 3 Schedule

- **See course website!**

Project 3 Overview

- **Goal:** Add support for preemptive scheduling and synchronization to the kernel
- Read the project spec for details
- Starter code can be found on the lab machines (/u/318/code/project3)
- **Start early**

Project 3 Overview

- The project is divided into three parts:
  - Timer interrupt/preemptive scheduling
  - Blocking sleep
  - Synchronization primitives
- Get each part working before starting the next
- Use test programs provided to test each part:
  - Use the script **settest** to set the test you'd like to use

Project 3 Overview

1. **Preemptive Scheduling**:
   - Implement timer interrupt in **entry.S**
2. **Blocking Sleep:**
   - Implement in **scheduler.c**
3. **Synchronization Primitives:**
   - Implement in **sync.c** and **sync.h**
   - Implement condition variables, semaphores, and barriers
   - How to implement them free of race conditions?

Project 3 Test Programs

- Five test programs are provided for your convenience
- Preemptive scheduling:
  - `test_regs` and `test_preempt`
- Blocking sleep:
  - `test_blocksleep`
- Synchronization primitives:
  - `Test_barrier` and `test_all` (tests everything)
- You are more than welcome to create your own tests!

Preemptive Scheduling

Once a process is scheduled, how does the OS regain control of the processor?

Preemptive Scheduling

- Tasks are preempted via timer interrupt IRQ0
- A time slice determines when to preempt (`time_elapsed` variable in **scheduler.c**)
- IRQ0 increments the time slice in each call
- Round-robin scheduling:
  - Have one task running and the others in queue waiting
  - Save the current task (context switch) before preempting
  - Change the current running task to next task in queue

Timer Interrupt

- Tasks are preempted via timer interrupt IRQ0
- Interrupts are labeled by their interrupt request numbers (IRQ):
    - An IRQ number corresponds to a pin on the programmable interrupt controller (PIC)
    - The PIC is a chip that manages interrupts between devices and the processor
- When receiving an interrupt, how does the processor know where to jump to?

Interrupt Initialization

- The OS needs to initialize a table of addresses to jump to for handling different interrupts
- In this project, the interrupt descriptor table (IDT) is setup in `kernel.c:init_idt()`
    - Separate entry for each hardware interrupt
    - Separate entry for each software exception
    - One entry for all system calls
- Try to understand `init_idt()` and how the kernel services system calls in this assignment

Interrupt Handling

- What does the processor do on an interrupt?
  - Disables interrupts
  - Pushes the EFLAGS, CS and return IP in that order on the stack
  - Jumps to the interrupt handler
  - Reverses the process on the way out (`iret` instruction)
- In this assignment, you will implement the IRQ0 handler

Implementing the IRQ0 Handler

- Send an “end of interrupt” to the PIC
  - Allows the hardware to deliver new interrupts
- Increment the number of ticks, a kernel variable (`time_elapsed`) for keeping track of the number of timer interrupts
- Increment `entry.S:disable_count`:
  - A global kernel “lock” for critical sections
  - Call ENTER_CRITICAL to increment (use ENTER_CRITICAL only when interrupts are disabled!)

Implementing the IRQ0 Handler

- If the current running task is in “user mode,” make it `yield()` the processor
  - Use the `nested_count` field of the PCB to check this
- If in kernel thread or kernel context of user process, let it continue running
- Decrement `entry.S:disable_count` using LEAVE_CRITICAL
- Return control to the process using `iret`

Watch Out For...

- **Safety:** When accessing kernel data structures, prevent race conditions by turning interrupts off
  - Use `enter_critical()` and `leave_critical()` for critical sections
- **Liveness:** Interrupts should be on most of the time
- You need to carefully keep track of the sections of code where interrupts are enabled/disabled

Sleep + Synchronization

Implementing Sleep - Busy Wait?

- Starter code implements sleep w/ while loop
- What's the problem with busy sleeping?

Implementing Sleep - Busy Wait?

- Starter code implements sleep w/ while loop
- What’s the problem with busy sleeping?
  - Wastes CPU Time
  - Even worse: if interrupts are disabled - halts the entire system!

Implementing Sleep - Blocking

- Use a new sleep queue
- Wake up process after n milliseconds
  - "Wake up" = put at end of ready queue
  - **`sleep(ms)`** guarantees that the process will be woken up no sooner than **`ms`** milliseconds, but potentially any time later.

Sleep - Things to think about

- Should interrupts be enabled or disabled?
- When should you try to wake up sleeping processes?
- What happens if all tasks are sleeping?

Synchronization Primitives

- Need to implement: condition variables, semaphores, and barriers
  - Lock implementation provided as reference
- Must work even with preemption:
  - Safety: Enable / Disable interrupts appropriately!
  - Liveness: Keep interrupts on as much as possible

Review: Condition Variables

- Properties:
  - Queue of threads waiting on condition to be true
- Operations:
  - Wait: block on condition + release lock while waiting
  - Signal: unblock one thread
  - Broadcast: unblock all waiting threads
  - (Threads must reclaim lock before running again)

Review: Semaphores

- Properties:
  - Number of “resources” available
  - Queue of waiting tasks
- Operations:
  - P / Down: decrement value + block if value < 0
  - V / Up: increment value + unblock one process

Review: Barriers

- Properties:
  - Number of tasks currently at barrier
  - Number of tasks required at barrier
  - Queue of waiting tasks
- Operations:
  - Wait: block if not all tasks have reached the barrier. Otherwise, unblock all waiting tasks

Tips + Other Notes

- Toughest part: handling when interrupts are enabled vs. disabled
  - Write helper functions as necessary
  - **`ASSERT`** is your friend!
- Review lecture slides (preemption, sync)

Design Review

- Be able to describe:
  - `irq0_entry`: workflow of the timer interrupt
  - **Blocking sleep**: how to sleep / wake up a task, how to handle special cases
  - **Sync Primitives**: what data structures to use + how to prevent race conditions
- Pseudocode is helpful

Questions?