Project 3
Preemptive Scheduler

COS 318
Fall 2015
Project 3: Preemptive Scheduler

- Goal: Add support for preemptive scheduling and synchronization to the kernel.
- Read the project spec for the details.
- Get a fresh copy of the start code from the lab machines. (/u/318/code/project3/)
- Start as early as you can and get as much done as possible by the design review.
Project 3: Schedule

- Design Review:
  - Sign up on the project page;
  - Please, draw pictures and write your idea down (1 piece of paper).

- Due date: Tuesday, 11/10, 11:55pm.
Project 3: Schedule

• Design Review:
  – Thursday, 10/29;
  – Answer the questions:
    ✓ **Irq0_entry**: The workflow of the timer interrupt;
    ✓ **Blocking sleep**:
      ✷ How do you make a task sleep and wake up?
      ✷ How do you handle the case when every task is sleeping and the ready queue is empty?
    ✓ **Synchronization primitives – condition variables, semaphores, barriers**:
      ✷ For each one, describe the data structure that you will use;
      ✷ The pseudo code for `condition_wait`, `semaphore_up`, `semaphore_down`, and `barrier_wait`. 
Project 3: Overview

• The project is divided into three phases:
  – Timer interrupt/preemptive scheduling;
  – Blocking sleep;
  – Synchronization primitives.
• Get each phase working before starting on the next one.
• Use provided test programs to test each component:
  – Use the script `settest` to set the test you want to use.
Project 3: Overview

• Implement preemptive scheduling:
  – Respond to timer interrupt: `entry.S`;
  – Blocking sleep: `scheduler.c`.

• Implement synchronization primitives: `sync.c` and `sync.h`:
  – What are the properties of condition variables, semaphores, and barriers?
  – How do you implement them free of race condition?

• Be careful: turn interrupts on/off properly:
  – Safety and liveness properties.
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, test_all (tests everything).
- Feel free to create your own test programs!
Pre-emptive scheduling

Once a process is scheduled, how does the OS regain control of the processor?
Pre-emptive 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 before preemption;
  - Change the current running task to the next one in the queue.
The timer interrupt

• Tasks are pre-empted through the timer interrupt:
  – Gives the OS the ability to decide on letting the current task continue.

• Interrupts are labeled by their interrupt request numbers (IRQ):
  – An IRQ number corresponds to a pin on the programmable interrupt controller (PIC);
  – PIC is a chip that manages interrupts between devices and the processor;
  – The timer corresponds to IRQ 0.

• 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 interrupts.
• In this project, the interrupt descriptor table (IDT) is setup in kernel.c:init_idt().
  – A separate entry for each hw interrupt;
  – A separate entry for each sw exception;
  – One entry for all system calls.
• You are encouraged to understand init_idt() and how the kernel services system calls in this project.
Interrupt handling

• What does the processor do on an interrupt?
  – Disable interrupts;
  – Push the flags, CS and return IP in that order on the stack;
  – Jump to the interrupt handler;
  – Reverse the process on the way out (iret instruction).

• In this project, you will implement the IRQ 0 handler:
  • Crucial for a pre-emptive scheduling OS.
Implementing the IRQ 0 handler

- Send an “end of interrupt” to the PIC:
  - Allow the hw to deliver new interrupts.
- Increment the number of ticks, a kernel variable (time_elapsed) for keeping track of the number of timer interrupts:
  - Timer initialized so that each tick corresponds to 1ms in real time.
- 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 IRQ 0 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:** interrupt should be on most of the time.
- **You need to carefully keep track of the sections of code where interrupts are enabled/disabled.**
Implement process sleep()

• Option 1: Busy sleeping
  – Template code (schedler.c:do_sleep()) has a “busy-wait” version of sleep, where the kernel uses a while loop.

• What’s the problem with option 1?

• How to implement a “blocking” version of sleep()?
Implement process sleep()

- Option 2: Blocking sleep
  - Use your own “sleep queue:”
    - This is not the ready queue;
  - Do the timing using number of ticks;
  - Wake up a process when the number of ticks reaches a specific value:
    - sleep(ms) guarantees that the process will be waken up no sooner than ms milliseconds, but it can potentially be any time later.
  - Carefully handle the case when all tasks are sleeping!
  - When does the kernel try to wake up sleeping processes?
Synchronization primitives

- Implement condition variables, semaphores, and barriers:
  - An implementation of locks is available.
- You need to design the required data structures and implement the primitives.
- All your primitives must work correctly on the face of pre-emption:
  - Safety: Turn interrupts on and off properly!
  - Liveness: Keep interrupts on as much as possible.
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 and release the mutex while waiting;
  – Signal: unblock once condition is true.

• Broadcast operation notifies all waiting threads.

• Refer to the slides of the 10/7 lecture
Review: semaphores

• Properties:
  – Control access to a common resource;
  – A value keeps track of the number of units of a resource that is currently available;
  – Queue of processes that are waiting.

• Two main operations:
  – Down: decrement value and block the process if the decremented value is less than zero;
  – Up: increment value and unblock one waiting process.

• Refer to the slides of the 10/7 lecture
Review: barriers

• Properties:
  – A barrier for a group of tasks is a location in code at which each task of the group must stop until all other tasks reach the barrier;
  – Keep track of the number of threads at barrier and the number of threads running;
  – Maintain queue of processes that are waiting.

• Main operations:
  – Wait: block the task if not all the tasks have reached the barrier. Otherwise, unblock all.

• Refer to the slides of the 10/7 lecture