COS 318   PROJECT 2
NON-PREEMPTIVE SCHEDULING
Outline

- Project is due at 23:59 on October 13
- Design reviews are 19:00 - 22:00 on Oct 6. Sign up!
- Today: go through the project, get you started
- Next time: design review summary, Q/A
Overview

* Target: Building a kernel that can switch between executing different tasks (task = process or kernel thread)

* Read the spec on course website

* Your grade will be determined partly on whether you handle subtle issues correctly. So don’t overlook any aspect.
What you need to deal with?

- Process Control Block (PCB)
- Context switching procedure
- System call mechanism
- Stacks
- Mutual Exclusion
Process Control Block

* kernel.h
* What should be in PCB?
  * pid, stack?
  * next, previous?
* What else?
Processes Example

COS 318
  go_to_class();
  yield();
  go_to_precept();
  yield();
  design_review();
  yield();
  coding();
  exit();

Life
  have_fun();
  yield();
  play();
  yield();
  do_stuff();
  yield();
  ......
Control Flow

COS 318
- go_to_class();
- yield();
- go_to_precept();
- yield();
- design_review();
- yield();
- coding();
- exit();

Real life
- have_fun();
- yield();
- play();
- yield();
- do_stuff();
- yield();
- ......
What is yield()?

* yield(): switch to another task
* For a task itself, it is a normal function call:
  * Push a return address on the stack
  * transfer control to yield()
* yield():
  * do_stuff();
  * return
* Task calling yield() has no knowledge of what do_stuff() is
Isolation

* Task must have their own:
  * registers
  * stack
  * ...... (for future assignments)
* Two techniques to achieve isolation
  * Division in space: allocate separate resources
  * Division in time: save and restore contexts
* Which one apply here?
Stack and Registers

- Allocate separate stacks in _start()
- yield():
  - save registers, including %esp
  - do_stuff()
  - restore registers
  - return
- Where are registers stored?
  - In the process control block (PCB)
The Secret Business Plan

COS 318

go_to_class ();
    yield();
    
    design_review();
    yield();
    
    yield returns

Real life

yield returns

have_fun();
    yield();
    
    do_stuff();
    yield();

overlapped calls?
No, they are not

* yield() calls appear to be overlapped
* Yet yield returns immediately to a different task, not the one that calls it
* Secret plan of yield()?
  * save registers
  * find the next task T
  * restore that task T’s saved registers
  * return to task T
Find the Next Task

- The kernel must keep track of which tasks have not exited yet
- The kernel should run the task that has been inactive for long
- What is the natural data structure?
- Please explain your design in the design review
Threads and Processes

* To yield, requires access to the scheduler’s data structures
* Kernel threads have access
  * scheduler.c : do_yield()
* User processes should not, but do for this project temporarily
* How should they get access?
To make a system call, a process:

- pushes the call number and arguments onto its stack
- interrupt/trap mechanism (later assignment), which elevates privileges and jumps into the kernel in a controlled manner
- In his project, processes have elevated privileges all the time
- Two system calls: yield() and exit()
entry.s: kernel_entry()

- kernel.c:
  - _start() stores the address of kernel_entry() at ENTRY_POINT (0xf00)
- Processes make system calls by:
  - loading the address of kernel_entry from 0xf00
  - passing the system call number to kernel_entry
- kernel_entry must save the registers and switch to the kernel stack, and reverse the process on the way out
Kernel and User Stack

- Processes have two stacks
  - user stack: for process to use
  - kernel stack: for kernel to use when executing system calls on behalf of the process
- Kernel thread has only one: kernel stack
- Suggestion: put them in memory 0x10000 - 0x20000
  - 4kb stack should be enough
  - upper limit = 640k (0xa000)
Memory Layout

- Entry Point: 0xf00
- PCBs: 0x1000
- Proc 1: 0x4000
- Proc 2: 0x7000
- Kernel Stack: 0x9000
- Proc 1's Kernel Stack: 0x10000
- Proc 1's User Stack: 0x11000
- Proc 2's Kernel Stack: 0x12000
- Proc 2's User Stack: 0x13000
Mutual Exclusion

* The calls available to threads are
  * `lock_init(lock_t *)`
  * `lock_acquire(lock_t *)`: check lock, block itself if cannot get it
  * `lock_release(lock_t *)`
* The precise semantics we want are described in the project spec
* There is exactly one correct trace
Timing a Context Switch

* `util.c : get_timer()` returns the number of cycles since boot

* There is only one process for your timing code, but it is given twice in tasks.c
  
  * use a global variable to distinguish the first execution from the second
Design Review Requirement

* Sign up for 10 minutes meeting with TA on project website
* Data structure design
* Context switching
* system calls design
* mutual exclusion design
* Please draw pictures and write your idea down (1 piece of paper)
* See project website for more details
QUESTIONS?