COS 318 Project 2
Pre-emptive scheduling
Administrivia

• Contact information
  • Srinivas Narayana, or “NG” if you like that.
  • narayana@(cs.)princeton.edu
  • COS 314
  • Friday 2 – 4 pm at the fishbowl

• Design reviews
  • Oct 14, Oct 15. 2-4 pm at the fishbowl
  • Sign up forms up on the project page
  • Please draw pictures and write your idea down (1 piece of paper)

• Project due Wednesday Oct 19 at noon!
Project 2 overview

- Target: Building a kernel that can switch between executing different tasks (task = process or kernel thread) in a non-preemptive fashion.

- Read the project spec for complete details.

- Subtle aspects are important.
  - “God lives in the details.”
What you need to deal with

- Process control blocks (PCB)
- User and kernel stacks
- Context switching procedure
- Basic system call mechanism
- Mutual exclusion
Assumptions for this project

- Processes run under elevated privileges

- Non-preemptible tasks
  - Run until they voluntarily yield or exit

- Fixed number of tasks
  - Allocate per-task state statically in your program
Process Control Block

- Definition in kernel.h
- What is its purpose?
- What should be in the PCB?
  - Pid
  - Stack segment information
  - Next, previous?
  - What else?
Task scheduling example

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

Life:
have_fun();
yield();
play();
yield();
do_random_stuff();
yield();
...


Control Flow

COS 318:

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

Life:

have_fun();
yield();
play();
yield();
do_random_stuff();
yield();
...

What is yield()?

- Switch to another task
- For a task itself, it's a normal function call
  - Push a return address (EIP) on the stack
  - Transfer control to yield()
- The task calling yield() has no knowledge of what yield() does
- yield():
  - Need to save and restore process state
What is this “process state”?

- When a task resumes control of CPU, it shouldn't have to care what transpired in the meantime.

- What should you do to give the task this abstraction?
yield(): stack and registers

- Allocate separate stacks for tasks in kernel.c: _start()

- yield() should:
  - Save general purpose registers (%eax, ..., including %esp)
  - Save flags
  - Instruction pointer?

- Where do you save these things?
  - PCB

- When does yield() return?
Who does yield() return to?

• Yield() returns immediately to a different task, not the one that calls it!

• Agenda of yield():
  • Save current task state
  • Pick the next task T to run
  • Restore T's saved state
  • Return to task T!

• You just executed a context switch!
Finding the next task

- The kernel must keep track of who hasn't exited yet
- Run the task that has been inactive for the longest.
- What's the natural data structure?
  - Please explain your design in the design review!
Calling yield()

- To call yield(), a process needs the addresses of the functions and be able to access these addresses.
- Kernel threads: no problem!
  - Scheduler.c: do_yield()
- User processes: should not have direct access
  - But in this project, processes run at kernel privileges
  - Now, how to get access?
System calls

• yield() is an example of a system call.
• To make a system call, typically a process:
  • Pushes a system call “number” and its arguments onto the stack
  • Uses an interrupt/trap mechanism to elevate privileges and jump into the kernel
• In this project though, processes have elevated privileges all the time.
• 2 system calls: yield() and exit()
Jumping into the kernel:
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 ENTRY_POINT
  - Calling the function at this address with a system call number as an argument

- kernel_entry(syscall_no) must save the registers and switch to the kernel stack, and reverse the process on the way out.
Allocating stacks

- Processes have two stacks
  - User stack: for the process to use
  - Kernel stack: for the kernel to use when executing system calls on its behalf
- Kernel threads need only one stack.
- Suggestion: Put them in memory 0x10000-0x20000.
  - 4kb stack should be enough.
Memory layout

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

- Lock-based synchronization is related to process scheduling.
- The calls available to threads are
  - `lock_init(lock_t *)`
  - `lock_acquire(lock_t *)`
  - `lock_release(lock_t *)`
- Precise semantics we want are described in the spec.
- There is exactly one correct trace.
Timing a context switch

- **util.c**: `get_timer()` returns 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.
Questions?
Think about...

- What should you do to jump to a kernel thread for the first time?
  - Process?
- How to save stuff into the PCB? In what order?
- Code up and test incrementally
  - Most effort spent in debugging, so keep it simple
- Start early
  - Plenty of tricky bits in this assignment
  - Do move past the design review by Friday!