Project 2
Non-preemptive Kernel

COS 318

Fall 2015

Project 2: Schedule

- Design Review:
  - Monday, 10/12;
  - Answer the questions:
    - ✓ **Process Control Block:** What will be in your PCB and what will it be initialized to?
    - ✓ **Context Switching:** How will you save and restore a task's context? Should anything special be done for the first task?
    - ✓ **Processes:** What, if any, are the differences between threads and processes and how they are handled?
    - ✓ **Mutual Exclusion:** What's your plan for implementing mutual exclusion?
    - ✓ **Scheduling:** Look at the project web page for an execution example.

Project 2: Schedule

- Design Review:
  - Sign up on the project page;
  - Please, draw pictures and write your idea down (1 piece of paper).
- Due date: Sunday, 10/18, 11:55pm.

Project 2: Overview

- Goal: Build a non-preemptive kernel that can switch between different tasks (task = process or kernel thread).
- Read the project spec for more details.
- Start early.

What is a non-preemptive kernel?

COS 318:

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

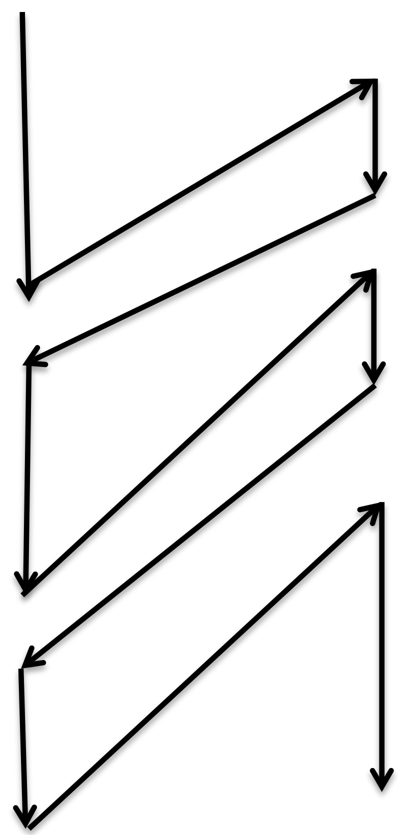
Life:

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

What is a non-preemptive kernel?

COS 318:

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



Life:

```
have_fun();
yield();
play();
yield();
do_random_stuff()
yield();
…
```

What you need to deal with

- Process control blocks (PCB).
- User and kernel stack.
- 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 info?
  - Next, previous?
  - What else?

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 the process state?

- When a task resumes control of CPU, it shouldn't have to care about what happened when it was not running.
- 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?

Where 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.

Finding the next task

- The kernel must keep track of which tasks have not exited yet.
- Run the task that has been inactive for the longest time.
- 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 kernel.
- In this project though, processes have elevated privileges all the time.
- Two 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 behalf of the process.
- Kernel threads need only one stack.
- Suggestion: put them in memory 0x40000-0x52000:
  - 4kb for the stack should be enough.

Memory layout

0x00000	BIOS
0x00F00	ENTRY_POINT
0x01000	Kernel
0x10000	Process 1
0x20000	Process 2
0x30000	Process 3
0x40000	Stacks
0x9FFFE	Kernel Stack (set by bootblock.s)
0xA0000	Video RAM
	…

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 project specification.
- 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.

Things to thing about…

- What should you do to jump to a kernel thread for the first time?
- How to save CPU state 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.