Project 2
Non-preemptive Kernel

COS 318

Fall 2016

Project 2: Schedule

- Design Review:
  - Monday, 10/10;
  - 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: Tuesday, 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?

- Current running task will lose its CPU or running state:
  - 1. yield
    2. block: I/O operation; lock (thread)
  - 3. exit

What is a non-preemptive kernel?

COS 318:

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

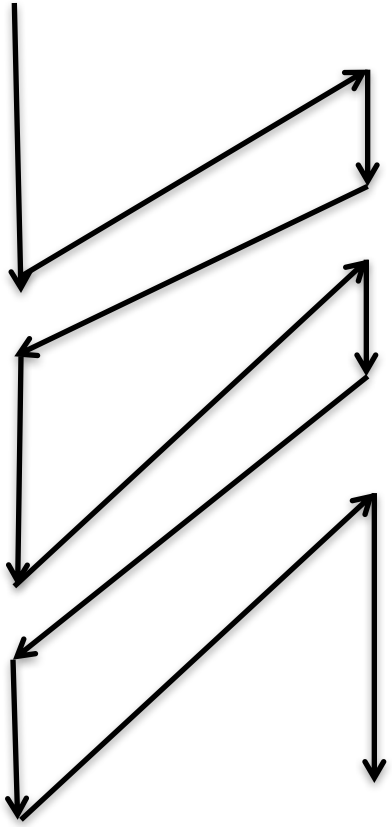
Life:

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

What is a non-preemptive kernel?

COS 318:

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



Life:

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

What you need to deal with

- 1. Process control blocks (PCB).
- 2. User and kernel stack.
- 3. Context switching procedure.
- 4. Basic system call mechanism.
- 5. Mutual exclusion.

Assumptions for this project

- Protected Mode -> don't worry about segment register
- Non-preemptive tasks:
  - run until they yield, exit or block.
- Fixed number of tasks:
  - allocate per-task state(PCB) statically in your program.
  - Each task's stack size is fixed.

1. Process Control Block

- Defined in kernel.h. and should be Initialized in kernel.c:_start();
- What is its purpose?
- What should be in the PCB?
  - PID
  - Stack info?
  - All registers?
  - What else?

2. Allocating stacks

- Allocate separate stacks for tasks in kernel.c: _start()
- Processes have two stacks(theoratically):
  - user stack – for the process to use;
  - kernel stack – for the kernel to use when executing system calls on behalf of the process.
  - **Option: only use one stack!**
- Kernel threads need only one stack.
- Suggestion: put them in memory 0x40000-0x52000:
  - 4kb for the stack should be enough.

3. Context Switch

- Implement a queue structure first!!!!
- Ready queue: Contains all PCBs or addresses of all PCBs of ready tasks.
- Blocked queue:
  - When do you need to push or pop your task?

3. Context Switch

- 1. Save state of task into the PCB. (Optional)
- 2. Push the current PCB into the ready queue or block queue. (Optional)
- 3. Pop the PCB of new task from ready queue.
- 4. Restore the state of new task.
- 5. Start new task.

3. Context Switch
Save state of tasks

- When a task resumes control of CPU, it shouldn't have to care about what happened when it was not running.
  - save general purpose registers (%eax, …, including %esp);
  - save flags.
  - instruction pointer?
- Where do you save these things?
  - PCB.

3. 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!

3. Calling yield()

- To call yield(), a task 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.
  - Now, how to get access?

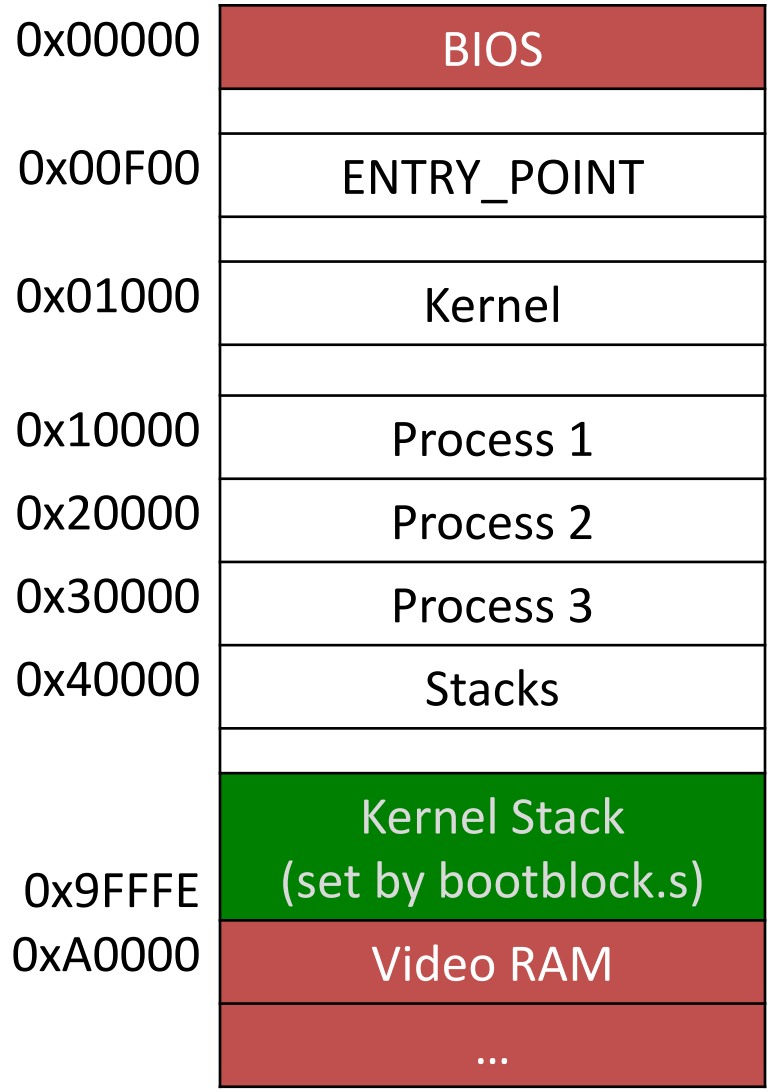
4. System calls

- 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.
- Two system calls: yield() and exit().

4. 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.

Memory layout



5. Mutual exclusion through locks

- Lock-based synchronization is related to thread 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.