Lab 2 - Non-Preemptive Kernel

Operating Systems, CS-UH 3010

Assigned:	Sep 20, 2022	Due:	Oct 10, 2022

Do not share the start codes or virtual image provided in this lab, as they were kindly provided by Princeton without permission to redistribute.

This assignment is more complex and harder than the first but if you do it, things that we've been discussing in class will all click and fall into place. It also includes a bit of assembly but hey by now you are almost pros. **START NOW!**

1 Overview

With the bootup mechanism from the first project, we can start developing an operating system kernel. The goal of this project is to design and implement a (a) simple multiprogramming kernel with a (b) non-preemptive scheduler. Although the kernel will be very primitive, you will be dealing with several core mechanisms and applying techniques that are important for building advanced, multiprogrammed operating systems.

The kernel of this project and future projects supports both user processes and kernel threads. Note that the processes in this project are different from those in traditional operating systems (like Unix). Our processes have their own protection domains, but they share a flat address space and (for the moment) run in kernel mode. In the future, we will be providing processes with separate address spaces. Our threads are kernel level threads that always share the same address space with the kernel; they will never run in user mode.

The kernel threads are linked together with the kernel, so they can access kernel data and functions directly. On the other hand, processes use the single entry-point mechanism described in the lecture. From now on, the kernel will use the protected mode of the x86 processors instead of the real mode. In this project, everything runs in kernel mode. You are required to do the following:

1. Initialize processes and threads including their stack(s) and related kernel data structures.
2. Implement a simple, non-preemptive scheduler such that every time a thread calls yield() or exit(), the scheduler picks the next thread and schedules it. As discussed in class, threads need to explicitly call yield() or exit(), in order to invoke the scheduler, otherwise a thread can run forever.
3. Implement a very simple system call mechanism. Since everything in this project runs in only the kernel mode, the system call mechanism can be a simple jump table. But the basic single-entry system call framework should be in place. You are required to implement two system calls: yield() and exit(). As previously mentioned, processes can make system calls, while threads can call scheduler functions directly.

4. Implement two synchronization primitives for mutual exclusion: `lock_acquire()` and `lock_release()`. Their semantics should be identical to that described in Birrell's paper (see BirrellThreads Paper). To implement `lock_acquire()` and `lock_release()`, the scheduler needs to support `block()` and `unblock()` for threads and processes. Blocking and unblocking is performed within the kernel - these are not system calls.

5. Measure the running time of a process/thread context switch.

We provide you with 23 files. Don't be scared by so many files since you are already familiar with some of them; you will need to change only 8 of them (listed in **bold face**).

1. Makefile: A make file for you to compile with the right options.
2. bootblock.S: Code for the bootblock that handles arbitrary size image files, switches the machine into protected-mode, and creates a flat address space.
3. createimage.c: Your familiar Linux tool to create a bootable operating system image on the USB disk.
4. **kernel.h**: Interface for the kernel.
5. **kernel.c**: A template for you to write the implementation of the kernel.
6. common.h: Commonly used definitions.
7. **lock.h**: Interface for the synchronization primitives.
8. **lock.c**: Template for you to write the implementation of the synchronization primitives.
9. th.h: Declaration of threads in th1.c and th2.c.
10. th1.c: Code of the first kernel thread.
11. th2.c: Code of two kernel threads to test the synchronization primitives.
12. **th3.c**: Code to time context switch for threads.
13. process1.c: A user process.
14. process2.c: Another user process.
15. **process3.c**: Code to time context switch for processes.
16. scheduler.h: Interface of the scheduler, part of the kernel
17. **scheduler.c**: Template for you to write scheduler functions
18. **entry.S**: Template for assembly code that used by the kernel and scheduler
19. util.h: Declaration of useful routines like `print_int()` and `get_timer()`.
20. util.c: Implementation of the functions declared in util.h.
21. syslib.h: Interface for the kernel functions available to processes.
22. syslib.c: Implementation of the functions declared in syslib.h.
23. bochsrc: The usual bochs initialization file.

2 Detailed Requirements

You need to understand the PCB (Process Control Block) and decide whether you would like to add additional items for your needs. The PCB provided has essential fields for a context switch. Since the processes and threads share the same address space in this project, the PCB can be very simple. Consider how data is saved during a context switch, and which stacks are used by threads and processes.

The bootblock given for this assignment not only loads your kernel but also sets up the memory system, by initializing the Global Descriptor Table and setting up the segment registers so that you can refer to addresses directly in a flat 1 MB address space. You should probably not modify the segment registers or GDT if you want the processes to keep working.

2.1 Kernel Threads & Scheduling

The kernel proper consists of a collection of kernel threads. These threads run in the most privileged protection ring, and their code is part of the kernel loaded by the boot block. Kernel threads share a flat address space, but they do not share register values or stacks.

After the boot block has loaded the kernel, it calls the function `kernel.c:_start()`. This function performs the setup necessary to run the tasks listed in tasks.c by initializing process control blocks (PCBs) and allocating a kernel stack for each task, plus a user stack for processes. You are encouraged to allocate space for the PCBs statically with `pcb_t pcbs[NUM_TASKS]`. Make sure to loop on `NUM_TASKS` when setting up PCBs as the number of tasks may be different during testing.

Note that in this project, processes need twice as much context information as threads. It's OK to use the same PCB structure for both processes and threads, and just waste space on threads by letting the user context information go unused. Stacks are `STACK_SIZE` bytes and are allocated contiguously at increasing addresses starting with `STACK_MIN`. Use the macro `common.h:ASSERT([assertion])` to enforce the invariant that all stacks lie in the interval `STACK_MIN` exclusive to `STACK_MAX` inclusive.

Since you are writing a non-preemptive scheduler, kernel threads run without interruption until they yield or exit by calling `scheduler.c:do_yield()` or `scheduler.c:do_exit()`. These functions in turn call `entry.S:scheduler_entry()`, which saves the contents of the general purpose and flags registers in the PCB of the task that was just running. `scheduler.c:scheduler()` is then called to choose the next task, restore the next task's registers, and return to it. `scheduler.c:scheduler()` chooses the next task to run in a strict round-robin fashion, and in the first round, the tasks are run in the order in which they are specified in tasks.c.

`scheduler.c:scheduler()` **increments the global variable** `scheduler.c:scheduler_count` **every time it runs. Do not change this.**

2.2 Processes and System Calls

In future projects, processes will have their own protected address spaces and run in a less privileged protection ring than the kernel. For now, however, there are only slight differences between threads and processes. Unlike threads, processes are allocated two stacks: one user stack, for the process proper, and one kernel stack, for system calls made by the process. Moreover, process code is not linked with the kernel, although for this project it is part of the image loaded by the boot block.

There are two system calls: `syslib.S:yield()` and `syslib.S:exit()`, which processes call. These correspond in principle to the calls `do_yield()` and `do_exit()` that the threads use. Processes cannot invoke `do_yield()` and `do_exit()` directly though because they don't know the address of these functions, and in later projects they won't have the requisite privileges to modify the PCBs.

This project uses a dispatch mechanism to overcome the difficulty of processes not knowing the addresses of `do_yield()` and `do_exit()`. kernel.h defines a spot in memory called `ENTRY_POINT` at which `_start()` writes the address of the function label kernel_entry (in entry.S) at run-time. When a process calls `yield()` or `exit()`

(in syslib.S), `SYSCALL(i)` (in the same file) gets called with an argument that represents which system call the caller wants (yield or exit), similar to how a number was stored in `%ah` before initiating a BIOS interrupt. `SYSCALL(i)` in turn calls `kernel_entry`, whose address was saved at `ENTRY_POINT`. `kernel_entry` saves the process's context to its PCB and calls `kernel_entry_helper()`, which then calls the appropriate function that was 'unreachable' from the process before: `do_yield()` or `do_exit()`. When `kernel_entry` is returned to when the process gets run again, it has got to restore the process's context.

2.3 Mutual Exclusion

lock.c contains three functions for use by kernel threads: `lock_init(l)`, `lock_acquire(l)`, and `lock_release(l)`. `lock_init(l)` initializes lock l; initially no thread holds the specified lock. In the event an uninitialized lock structure is passed to the other functions, the results are up to you. `lock_acquire(l)` must not block the thread that called it when no other thread is holding is holding l.

Threads call `lock_acquire(l)` to acquire lock l. If no thread currently holds l, it marks the lock as being held and returns. Otherwise it uses `scheduler.c:block()` to indicate to the scheduler that the thread is now waiting for the lock and should be put on the blocked queue. If there are one or more threads on the blocked queue when `lock_release(l)` is called by the thread holding the lock, the lock remains locked and the thread at the head of the blocked queue is put on the ready queue at the back, through a regular enqueue operation. If no threads are waiting, it marks the lock as being not held and returns. The results of a thread attempting to acquire a lock it already holds or to release a lock it doesn't hold are up to you.

We have put a non-conforming implementation of mutual exclusion in lock.c so that you can write and test the kernel threads part in isolation. When you are ready to test your own implementation, change the constant SPIN in lock.c to FALSE.

Note: your lecture notes describe how to synchronize access to the fields associated with a lock. Since there is no preemption, if `lock_acquire(l)` and `block()` don't call `do_yield()` in the middle of manipulating data structures, you don't need to worry about race conditions while these functions do their job. In particular, it is not necessary to protect the scheduler data structures with a test-and-set lock.

2.4 Timing a Context Switch

The function `util.c:get_timer()` returns the number of instructions that have been executed since boot. Using `util.c:print_int()` and `util.c:print_str()`, use the first line of the display to show the number of cycles required by a context switch between threads and between processes. We've left space in th3.c and process3.c for your code.

Please turn in your measurement code and have it run when I compile and boot your OS.

3 Design Review

For the design review, you need to have figured out all the details including the PCB data structure, the call sequence executed during a context switch or system call, how process state is saved and restored, etc.

In other words, you should understand the role of each function, being ready to implement it. Before you come to meet Nabil, you should prepare for a brief 10 min presentation on a piece of paper or a text file. An oral-only presentation is not acceptable.

The design review is worth 5 points! So come prepared.

4 Hints

- Use a good IDE like Eclipse C/C++ when writing your code. It makes your code readable and avoids many mistakes due to incorrect filename, function names, etc.
- The boot loader provided to you for this project does more than the one you wrote last project. Specifically, it enables address line 20 and enters protected mode.
- One of the first things you will need to do is to implement a queue data structure. gdb is a much nicer environment than bochs in which to debug, so we strongly recommend you that you test your data structure code in user-space first.
- We've placed examples of how to call C from assembly and vice versa in `hello1.c` and `hello2.S`. Build the executable by typing make hello.
- The dependencies of the parts of this project look like this:
  - Processes $\rightarrow$ Timing a process context switch
  - Kernel threads $\rightarrow$ Mutual exclusion
  - Timing a thread context switch
- Saving and restoring the general purpose registers (eax, ecx, edx, ebx, esp, ebp, esi, edi) is reasonably straightforward, but the flags register (eflags) is tricky. The only way to save eflags is the instruction pushf, which pushes the contents of eflags onto the stack–eflags is not a valid operand for mov. Similarly, the only way to restore eflags is popf, which pops the top of the stack into eflags. Worse, you must take care not to alter any register before it is saved or after it is restored, and many instructions set and clear flags. Instructions that don't alter eflags include mov, pop, push, pushf, and ret, while arithmetic operations typically do. Consult the x86 manuals for more information.
- To allocate space for the PCBs, declare an array of PCB structures, for example, `static pcb_t pcbs[NUM_TASKS];` and then link them together as necessary.
- If you have an address as a `uint32_t` and want to write to it from C, you have to cast it to a pointer: `*(uint32_t *) address_as_an_integer = value;`
- The popf and pushf instructions do not take any operands. They always pop/push eflags from/onto the stack (i.e., the current value of esp, which may not correspond to any of the stacks you allocate to tasks).
- Adding `print_int` and `print_str` to your program can help you debug.
- Start from the simplest set of user programs:
  - First try only 1 task (changing the definition of tasks in task.c).
  - If your program works for this task, try another one.
  - If your program works for every individual task, try combination of many tasks.
  - Even for 1 task, you can comment out some codes which contain functions you're not ready to test. (For example, to test whether your program can start a user process, you can comment the calls to yield())
  - What helps is creating a control flow to help you understand how different components interact with each other. Here is a snippet of this flow, try to flush it out completely.

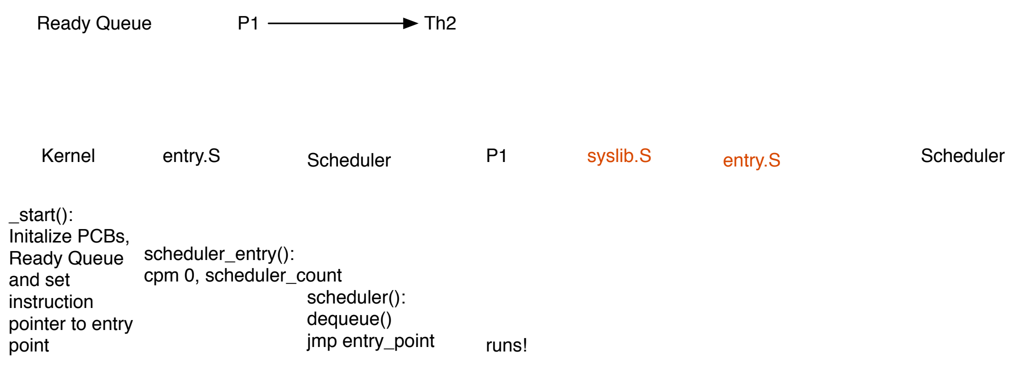
Ready Queue
P1
Th2
Kernel
entry.S
Scheduler
P1
syslib.S
entry.S
Scheduler
_start(): Initalize PCBs, Ready Queue and set instruction pointer to entry point
scheduler_entry(): cpm 0, scheduler_count
scheduler(): dequeue() jmp entry_point
runs!

Figure 1: Initial Partial Flow through modules as kernel switches from P1 to Thread 2.

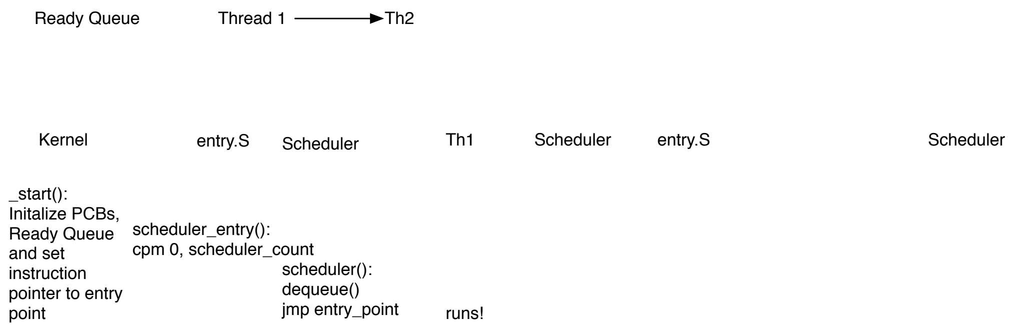
Ready Queue
Thread 1
Th2
Kernel
entry.S
Scheduler
Th1
Scheduler
entry.S
Scheduler
_start(): Initalize PCBs, Ready Queue and set instruction pointer to entry point
scheduler_entry(): cpm 0, scheduler_count
scheduler(): dequeue() jmp entry_point
runs!

Figure 2: Initial Partial Flow through modules as kernel switches from Thread 1 to Thread 2.

5 Definitions

address line 20	When PCs boot up, addresses `0x100000` to `0x1fffff` are aliases for `0x00000` to `0xfffff` to accommodate badly written real mode programs. Programs that use more than one megabyte of memory must disable this behavior. See `http://www.win.tue.nl/~aeb/linux/kbd/A20.html`.
kernel mode	Synonym for protection ring 0.
process control block	The process control block (PCB) contains all of the state the kernel needs to keep track of a process or kernel thread.
protected mode	In (16-bit) real mode, an address ds:si refers to byte 16*ds + si. In 32-bit protected mode, the segment registers contain offsets into the global descriptor table, whose entries control the translation of memory accesses into physical addresses. The boot block provided to you sets up the global descriptor table and segment registers so that no translation is applied; you can ignore the segment registers for this project.
protection ring	Intel processors since the 286 have four privilege levels, ring 0 through ring 3. Ring 0 is the most privileged; ring 3 is the least. Typically the kernel runs in ring 0, and user processes run in ring 3. For this project, all processes run in ring 0.
system call	In modern operating systems, processes are enjoined from accessing memory that they do not own. System calls are how processes communicate with the kernel. Typically, the process writes the system call number and its arguments in its own memory, and then it transfers control to the kernel with a special kind of far call. The kernel then performs the desired action and returns control to the process.
task	Following Linux, we use 'task' to mean process or kernel thread.

6 Grading Rubric

The project is worth 15 points. The breakdown is as follows:

Task	Deliverables	Points
Design Review	Correct descriptions of the main concepts of the project.	5
Start	OS can start a single task	2
Yield	`do_yield` and `yield` save and restore contexts correctly	3
Fifo	Without locks, scheduler obeys FIFO order	1
4 Threads	OS can unfold execution for the 4 threads in design review correctly	2
Mutex	Your lock can guarantee mutual exclusion and FIFO order for locks (that is, the thread acquiring the lock earlier gets it earlier)	2
Timing	th3.c and process3.c time JUST how long a `do_yield` or `yield` takes, respectively	1
All Tasks Load & Run	OS can successfully load and run all tasks provided in the start code	4
Poor Style	We reserve the right to remove as much as 1 point for submissions which are extremely confusing, obfuscated or over-complicated. Please write simple, readable code with comments. Provide a README file with your submission	-1
Bonus	Lets heat the competition: first group to submit a working lab gets a bonus point	+1

7 Submission

Please upload your group submission here: https://www.dropbox.com/request/uZgMWXawjUdH4RtdGmuh

8 Acknowledgments

Once again, we gratefully acknowledge the kindness of the staff of Princeton's COS 318 course for providing us with the material for (and most of the text of) this assignment.

Like always, best wishes and good luck!