Operating Systems (PG)
Assignment #3 - Pintos (Scheduling)
Deadline: 25th October 2018 11:55 PM

Pintos is a simple operating system framework for the 80x86 architecture. It supports kernel threads, loading and running user programs, and a file system, but it implements all of these in a very simple way. Pintos projects are run on a system simulator, that is, a program that simulates an 80x86 CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it. Using the QEMU simulator is recommended. This assignment requires a thorough understanding of the OS and creativity in designing your approaches to solve the problems.

It is strongly advised to go through all the links provided in detail as a lot of detailed specifications and ideas for your implementations are stated there.

Part1: Getting Started

Unlike tailored versions of commercial or open source OS such as Linux, Pintos is designed from the ground up from an educational perspective. Here is how to get started:

- Understand Pintos basics from reading material.
- Build the Pintos executable from source code and run it on a virtual machine.
- Get familiar with the source code (files and data structures).
- Links:
    - Introduction: http://www.stanford.edu/class/cs140/projects/pintos/pintos_1.html#SEC1
    - Source code: http://www.stanford.edu/class/cs140/projects/pintos/pintos.tar.gz
    - Setup/Installation: http://web.stanford.edu/class/cs140/projects/pintos/pintos_12.html
    - For building and running pintos follow sections 1.1.2 and 1.1.3 (Use QEMU or VMplayer): http://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html#SEC1
- Create a hello.c file which consists of main function and prints "Hello Pintos" message. Learn how to run this file inside pintos. For this you can look into the pintos/src/tests/threads. There are several test files present in this directory.

Pintos currently has a minimally functional threading system. Your job is to extend the functionality of this system to gain a better understanding of scheduling and synchronization problems. You will be working primarily in the threads directory for this assignment, with some work in the devices directory on the side. Compilation should be done in the threads directory.

Part2: Pre-emption of threads

Objective: The first step in the construction of a basic preemptive scheduler is the ability to preempt (that is, stop or pause) a currently scheduled task in favour of another task. The situation being modelled in this assignment is that of sleeping threads. When a thread is put to sleep for a certain duration, we would like another task to occupy the duration in between, rather than waiting for the thread to finish its sleep cycle and resuming its execution (busy waiting). In this part you must understand the thread sleep mechanism of pintos and provide an efficient solution to circumvent busy waiting.

Additional details: Reimplement timer_sleep(), defined in devices/timer.c. Although a working implementation is provided, it "busy waits," that is, it spins in a loop checking the current time and calling thread_yield() until enough time has gone by. Reimplement it to avoid busy waiting. Function:
`void timer_sleep (int64_t ticks)`

> Suspends execution of the calling thread until time has advanced by at least x timer ticks. Unless the system is otherwise idle, the thread need not wake up after exactly x ticks. Just put it on the ready queue after they have waited for the right amount of time.

timer_sleep() is useful for threads that operate in real-time, e.g. for blinking the cursor once per second.

All tests and related files are in pintos/src/tests. Please refer section 1.2.1 of the link below for verifying your output of test cases against expected output.
http://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html

Further Description of the task: Section 2.2.2
https://web.stanford.edu/class/cs140/projects/pintos/pintos_2.html#SEC15

Test cases that your implementation must pass will be shared on moodle.

Part3: Implementation of priority scheduling

Objective: Building on the previous part, this part requires you to implement a basic priority scheduling scheme. When a thread is added to the ready list that has a higher priority than the currently running thread, the current thread should immediately yield the processor to the new thread. Similarly, when threads are waiting for a lock, semaphore, or condition variable, the highest priority waiting thread should be awakened first. In the current scenario, priority value is taken but it has not been used anywhere for thread scheduling (See thread structure in src/threads/thread.h).

Additional details: Thread priorities range from PRI_MIN (0) to PRI_MAX (63). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. The initial thread priority is passed as an argument to thread_create(). If there's no reason to choose another priority, use PRI_DEFAULT (31).
A thread may raise or lower its own priority at any time, but lowering its priority such that it no longer has the highest priority must cause it to immediately yield the CPU. Implement the following functions that allow a thread to examine and modify its own priority. Skeletons for these functions are provided in threads/thread.c.

Function: `void thread_set_priority (int new_priority)`

> Sets the current thread's priority to new_priority. If the current thread no longer has the highest priority, yields.

Function: `int thread_get_priority (void)`

> Returns the current thread's priority.

Note: Priority donation is an interesting problem, but is **NOT** required as part of this assignment.

Further Description of the task: Section 2.2.3
https://web.stanford.edu/class/cs140/projects/pintos/pintos_2.html#SEC15

Test cases that your implementation must pass will be shared on moodle.

Part 4: Advanced Scheduler (MLFQS) (**Bonus**)

Objective: Implement a multilevel feedback queue scheduler similar to the BSD scheduler to reduce the average response time for running jobs on your system. Like the priority scheduler, the advanced scheduler chooses the thread to run based on priorities. Thus, we recommend that you have the priority scheduler working before you start work on the advanced scheduler. You must replicate the computational procedures taken by the BSD scheduler (calculating niceness, priority, recent_cpu and load_avg) in your implementation.

Additional details: This type of scheduler maintains several queues of ready-to-run threads, where each queue holds threads with a different priority. At any given time, the scheduler chooses a thread from the highest-priority nonempty queue. If the highest-priority queue contains multiple threads, then they run in "round robin" order.
You must write your code to allow us to choose a scheduling algorithm policy at Pintos startup time. By default, the priority scheduler must be active, but we must be able to choose the MLFQS scheduler with the -mlfqs kernel option. When the MLFQS scheduler is enabled, threads no longer directly control their own priorities. The priority argument to thread_create() should be ignored, as well as any calls to thread_set_priority(), and thread_get_priority() should return the thread's current priority as set by the scheduler.

Further Description of the task: Section 2.2.4
https://web.stanford.edu/class/cs140/projects/pintos/pintos_2.html#SEC15
BSD scheduler detailed working description:
https://web.stanford.edu/class/cs140/projects/pintos/pintos_7.html#SEC131

Test cases that your implementation must pass will be shared on moodle.

Upload Format:
Upload Format will be shared on the course moodle before the submission deadline. A detailed **report** of your implementation is expected with details such as implementation challenges, how you overcame them, strategies used and the reasoning behind them, etc.

Guidelines:
1. This is a group assignment. Students are to make groups of at most 2 people.
2. ZERO tolerance towards any kind of code plagiarism. Plagiarism will fetch you a ZERO. No negotiations.
3. DO NOT COPY/SHARE code/code-snippets (even a few lines of copied code would be detected and punished) - both the parties will get ZERO.