

Project - 1

CSE 421/521 – Operating Systems

Due: March 29th @11:59pm, 2018

1. Preparation

Before beginning your work, please read the following carefully:

- Chapters 3-5 from Silberschatz (9th Edition)
- Lecture slides on Processes, Threads and CPU Scheduling
- [Pintos Introduction](#)
- [Pintos Reference Guide](#) (*Sections referred below*)
- [Complete Pintos Documentation \(PDF file\)](#) -- for reference only

2. Task: Implement the Threading Component of Pintos OS

In this project, your team is asked to perform “kernel” level programming of the “Threading” component in the Pintos operation system. This project will help you to better understand threading, synchronization, and scheduling concepts.

Pintos is a simple operating system framework for the 80x86 architecture developed at Stanford University. It is considered as a successor of the Nachos instructional OS from UC-Berkeley. You will be given a functional OS with minimally functional thread system. Your job is to extend the functionality of this system to gain a better understanding of synchronization problems.

Before beginning this assignment, make sure you read these sections from Pintos Reference Guide: section [A.1 Pintos Loading](#), [A.2 Threads](#), [A.3 Synchronization](#), and [Appendix B \(4.4BSD Scheduler\)](#).

3. Setting Up The Pintos Environment

We have prepared a VM image of Pintos which will make the installation/configuration process much simpler for you. This will enable you to develop your pintos project on any Linux, Windows, or Mac system (i.e. your own machine) without much hassle. You can download the VM image at the following link: <http://ftp.cse.buffalo.edu/CSE421/UB-pintos.ova>

You will need VirtualBox software for your host OS to run this VM. You can download VirtualBox at the following link: <https://www.virtualbox.org/wiki/Downloads>

For the detailed instruction on how to setup this VM on your host OS, please see the "Guide for Pintos Installation on Your System" which is posted on Piazza: http://www.piazza.com/class/profile/get_resource/jcw8lxnkews2cb/jdavrjsmmmv5z9

In VirtualBox software, from the File menu, choose "Import Appliance" and point to the downloaded "UB-pintos.ova". You do not need to change any other settings for this procedure. Choose UB-Pintos from the left pane, and click on "Start", your virtual machine should now boot and get you to a login page, which requires the following username/password combination to login:

Username: os-class
Password: cse421521

To learn how to run, debug and test Pintos code, please read the [Pintos Introduction](#).

4. Implementation of the Project

You will be working primarily in the “threads” directory of the source tree for this assignment, with some work in the “devices” directory on the side. Compilation should be done in the “threads” directory.

4.1 Understanding Threads

The first step is to read and understand the code for the initial thread system. Pintos already implements thread creation and thread completion, a simple scheduler to switch between threads, and synchronization primitives (semaphores, locks, condition variables, and optimization barriers).

Some of this code might seem slightly mysterious. If you haven't already compiled and run the base system, as described in the introduction (see section [1. Introduction](#)), you should do so now. You can read through parts of the source code to see what's going on. If you like, you can add calls to `printf()` almost anywhere, then recompile and run to see what happens and in what order. You can also run the kernel in a debugger and set breakpoints at interesting spots, single-step through code and examine data, and so on.

When a thread is created, you are creating a new context to be scheduled. You provide a function to be run in this context as an argument to `thread_create()`. The first time the thread is scheduled and runs, it starts from the beginning of that function and executes in that context. When the function returns, the thread terminates. Each thread, therefore, acts like a mini-program running inside Pintos, with the function passed to `thread_create()` acting like `main()`.

At any given time, exactly one thread runs and the rest, if any, become inactive. The scheduler decides which thread to run next. (If no thread is ready to run at any given time, then the special “idle” thread, implemented in `idle()`, runs.) Synchronization primitives can force context switches when one thread needs to wait for another thread to do something.

The mechanics of a context switch are in “threads/switch.S”, which is 80x86 assembly code. (You don't have to understand it.) It saves the state of the currently running thread and restores the state of the thread we're switching to.

Using the GDB debugger, slowly trace through a context switch to see what happens (see section [E.5 GDB](#)). You can set a breakpoint on `schedule()` to start out, and then single-step from there. Be sure to keep track of each thread's address and state, and what procedures are on the call stack for each thread. You will notice that when one thread calls `switch_threads()`, another thread starts running, and the first thing the new thread does is to return from `switch_threads()`. You will understand the thread system once you understand why and how the `switch_threads()` that gets called is different from the `switch_threads()` that returns. See section [A.2.3 Thread Switching](#), for more information.

Warning: In Pintos, each thread is assigned a small, fixed-size execution stack just under 4 kB in size. The kernel tries to detect stack overflow, but it cannot do so perfectly. You may cause bizarre problems, such as mysterious kernel panics, if you declare large data structures as non-static local variables, e.g. “`int buf[1000];`”. Alternatives to stack allocation include the page allocator and the block allocator (see section [A.5 Memory Allocation](#)).

4.2 Source Files

For a brief overview of the files in the “threads/” directory, please see “[Section 2.1.2 Source Files](#)” in the Pintos Reference Guide. You will not need to modify most of this code, but the hope is that overviewing this section will give you a start on what code to look at. Also, please take a quick

look at the files in the “devices/” and “lib/” directories of the pintos source tree.

4.3 Synchronization

Proper synchronization is an important part of the solutions to these problems. Any synchronization problem can be easily solved by turning interrupts off: while interrupts are off, there is no concurrency, so there's no possibility for race conditions. Therefore, it's tempting to solve all synchronization problems this way, but **don't**. Instead, use semaphores, locks, and condition variables to solve the bulk of your synchronization problems. Read the tour section on synchronization (see section [A.3 Synchronization](#)) or the comments in “threads/synch.c” if you're unsure what synchronization primitives may be used in what situations. **Any implementation turning the interrupts off for synchronization purposes, will not get full credits!**

In the Pintos projects, the **only class of problem best solved by disabling interrupts** is coordinating data shared between a kernel thread and an interrupt handler. Because interrupt handlers can't sleep, they can't acquire locks. This means that data shared between kernel threads and an interrupt handler must be protected within a kernel thread by turning off interrupts.

This project only requires accessing a little bit of thread state from interrupt handlers. For the alarm clock, the timer interrupt needs to wake up sleeping threads. In the advanced scheduler, the timer interrupt needs to access a few global and per-thread variables. When you access these variables from kernel threads, you will need to disable interrupts to prevent the timer interrupt from interfering.

When you do turn off interrupts, take care to do so for the least amount of code possible, or you can end up losing important things such as timer ticks or input events. Turning off interrupts also increases the interrupt handling latency, which can make a machine feel sluggish if taken too far.

The synchronization primitives themselves in “synch.c” are implemented by disabling interrupts. You may need to increase the amount of code that runs with interrupts disabled here, but you should still try to keep it to a minimum.

Disabling interrupts can be useful for debugging, if you want to make sure that a section of code is not interrupted. You should remove debugging code before turning in your project. (Don't just comment it out, because that can make the code difficult to read.)

There should be no busy waiting in your submission. A tight loop that calls `thread_yield()` is one form of busy waiting.

4.4 Alarm Clock

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. Any implementation using busy-waiting will not get full credits!**

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.

The argument to `timer_sleep()` is expressed in timer ticks, not in milliseconds or any another unit. There are `TIMER_FREQ` timer ticks per second, where `TIMER_FREQ` is a macro defined in

devices/timer.h. The default value is 100. We don't recommend changing this value, because any change is likely to cause many of the tests to fail.

Separate functions `timer_msleep()`, `timer_usleep()`, and `timer_nsleep()` do exist for sleeping a specific number of milliseconds, microseconds, or nanoseconds, respectively, but these will call `timer_sleep()` automatically when necessary. You do not need to modify them.

If your delays seem too short or too long, reread the explanation of the “-r” option to pintos (see section [1.1.4 Debugging versus Testing](#)).

4.5 Priority Scheduler

Implement priority scheduling in Pintos. 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. 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.

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). The `PRI_` macros are defined in “threads/thread.h”, and you should not change their values.

One issue with priority scheduling is “priority inversion”. Consider high, medium, and low priority threads *H*, *M*, and *L*, respectively. If *H* needs to wait for *L* (for instance, for a lock held by *L*), and *M* is on the ready list, then *H* will never get the CPU because the low priority thread will not get any CPU time. A partial fix for this problem is for *H* to “donate” its priority to *L* while *L* is holding the lock, then recall the donation once *L* releases (and thus *H* acquires) the lock.

Implement priority donation. You will need to account for all different situations in which priority donation is required. Be sure to handle multiple donations, in which multiple priorities are donated to a single thread. You must also handle nested donation: if *H* is waiting on a lock that *M* holds and *M* is waiting on a lock that *L* holds, then both *M* and *L* should be boosted to *H*'s priority. If necessary, you may impose a reasonable limit on depth of nested priority donation, such as 8 levels.

You must implement priority donation for locks. You need not implement priority donation for the other Pintos synchronization constructs. You do need to implement priority scheduling in all cases.

Finally, 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. In the presence of priority donation, returns the higher (donated) priority.

You need not provide any interface to allow a thread to directly modify other threads' priorities.

4.6 Multilevel Feedback Queue Scheduler

Implement a multilevel feedback queue scheduler similar to the 4.4BSD scheduler to reduce the average response time for running jobs on your system. See section [B. 4.4BSD Scheduler](#), for detailed requirements.

Like the priority scheduler, the advanced scheduler chooses the thread to run based on priorities. However, the advanced scheduler does not do priority donation. Thus, we recommend that you have the priority scheduler working, except possibly for priority donation, before you start work on the advanced scheduler.

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 4.4BSD scheduler with the “-mlfq” kernel option. Passing this option sets `thread_mlfqs`, declared in “threads/thread.h”, to true when the options are parsed by `parse_options()`, which happens early in `main()`.

When the 4.4BSD 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.

5. Testing

Your project grade will be based on our tests. Each project has several tests, each of which has a name beginning with “tests”. To completely test your submission, invoke “make check” from the project “build” directory. This will build and run each test and print a “pass” or “fail” message for each one. When a test fails, make check also prints some details of the reason for failure. After running all the tests, make check also prints a summary of the test results.

You can also run individual tests one at a time. A given test *t* writes its output to “*t.output*”, then a script scores the output as “pass” or “fail” and writes the verdict to “*t.result*”. To run and grade a single test, make the “.result” file explicitly from the “build” directory, e.g. `make tests/threads/alarm-multiple.result`. If make says that the test result is up-to-date, but you want to re-run it anyway, either run `make clean` or delete the “.output” file by hand.

By default, each test provides feedback only at completion, not during its run. If you prefer, you can observe the progress of each test by specifying “VERBOSE=1” on the make command line, as in `make check VERBOSE=1`. You can also provide arbitrary options to the pintos run by the tests with “PINTOSOPTS='...'”, e.g. `make check PINTOSOPTS='-j 1'` to select a jitter value of 1 (see section [1.1.4 Debugging versus Testing](#)).

All of the tests and related files are in “pintos/src/tests”.

6. Design Document

Before you turn in your project, you must copy the [Project 1 Design Document Template \(threads.tmpl\)](#) into your source tree under the name “pintos/src/threads/DESIGNDOC” and fill it in. We recommend that you read the design document template before you start working on the project. See section [D. Project Documentation](#), for a sample design document that goes along with a fictitious project. You will need to decide and describe the main data structures, algorithms, and synchronization mechanisms that you are using / planning to use for each component of the project.

7. Grading

The grading of the project will be done according to the following rubric (93 points for the implementation + 12 points for the design document = TOTAL 105 points):

- (18 points) A completely working Alarm Clock implementation that passes all six (6) tests.
- (38 points) A fully functional Priority Scheduler that passes all twelve (12) tests.
- (37 points) A working advanced scheduler that passes all nine (9) tests.
- (12 points) A complete design document.

Run **“make grade”** to see how many total points the student receives from implementation (out of 93):

Functionality and robustness of alarm clock (tests/threads/Rubric.alarm):

4 / 4 tests/threads/alarm-single
4 / 4 tests/threads/alarm-multiple
4 / 4 tests/threads/alarm-simultaneous
4 / 4 tests/threads/alarm-priority
1 / 1 tests/threads/alarm-zero
1 / 1 tests/threads/alarm-negative

Functionality of priority scheduler (tests/threads/Rubric.priority):

3 / 3 tests/threads/priority-change
3 / 3 tests/threads/priority-preempt
3 / 3 tests/threads/priority-fifo
3 / 3 tests/threads/priority-sema
3 / 3 tests/threads/priority-condvar
3 / 3 tests/threads/priority-donate-one
3 / 3 tests/threads/priority-donate-multiple
3 / 3 tests/threads/priority-donate-multiple2
3 / 3 tests/threads/priority-donate-nest
5 / 5 tests/threads/priority-donate-chain
3 / 3 tests/threads/priority-donate-sema
3 / 3 tests/threads/priority-donate-lower

Functionality of advanced scheduler (tests/threads/Rubric.mlfqs):

5 / 5 tests/threads/mlfqs-load-1
5 / 5 tests/threads/mlfqs-load-60
3 / 3 tests/threads/mlfqs-load-avg
5 / 5 tests/threads/mlfqs-recent-1
5 / 5 tests/threads/mlfqs-fair-2
3 / 3 tests/threads/mlfqs-fair-20
4 / 4 tests/threads/mlfqs-nice-2
2 / 2 tests/threads/mlfqs-nice-10
5 / 5 tests/threads/mlfqs-block

- **CHECK CODE:** If Alarm clock implementation is based on “busy-waiting” (-14 points)
- **CHECK CODE:** If interrupts are turned off as the only synchronization mechanism (-14 points)

8. What to Submit?

1. You need to write a **design document** for your project as described in section 6. A softcopy of this design document (in PDF format) should be submitted three weeks before the project deadline.

The design document is due **March 8th @11:59pm.**

The submission of the design document should be done using your "submit_cse421" or "submit_cse521" script. (* NOTE: The submit scripts will only be used for the design document submission, not for the actual project source code submission. Please see below *).

2. You need to submit the complete **source tree** (all source files) of your project. The whole package should compile when the tester simply types **make** in the source code directory.

The project source code submission is due **March 29th@11:59pm.**

The submission of the project source code will be done using our department's **AutoLab** (<https://autograder.cse.buffalo.edu/>) system, which will allow you to see your grade even before submitting the project. The detailed instructions on this will be provided later.