CIS 520 - Spring 2018 Pintos Project #1 - Thread Programming and Synchronization

(Adapted from the Pintos Project by Ben Pfaff)

In this assignment, we give you a minimally functional thread system. Your job is to extend the functionality of this system to gain a better understanding of 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 as in Project #0. You can use your installation of Pintos from Project #0, or re-install Pintos (no need to re-install bochs).

DUE: Upload solution via K-State OnLine no later than 11:59 pm on Sunday, February 18, 2018.

TO DO: Upload a gzipped tar file called Proj1.tgz that includes:

- A design document (Design1.txt or Design1.pdf) in the pintos/src/threads directory.
- All of your modified Pintos source code; e.g., use the command:
  - `cd ~/cis520`
  - `tar czvf Proj1.tgz pintos`

1. Background

1.1 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. In Project #0, you already compiled and ran the base system. 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 executed 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 a 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 implemented in "threads/switch.S", which is 80x86 assembly code. (You don't have to understand it in detail.) 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. 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 functions 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. **Warning:** In Pintos, each thread is assigned a small, fixed-size execution stack that is 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.

1.2 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 or the comments in "threads/synch.c" if you're unsure what synchronization primitives may be used in what situations.

In the Pintos, 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 for very short amounts of time. 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.

1.3 Development Suggestions

It is a bad idea for each group member to work individually on his/her piece of code and then try to combine all of the code at the last minute to submit. We do not recommend this approach. Groups that use this approach often find that two changes conflict with each other, requiring lots of last-minute debugging. Some groups who have done this have turned in code that did not even compile or boot, much less pass any tests.

Instead, we recommend integrating your teams' changes early and often, using a source code control system such as Git. This is less likely to produce surprises, because everyone can see everyone else's code as it is written, instead of just when it is finished. These systems also make it possible to review changes and, when a change introduces a bug, drop back to working versions of code. You may want to use SourceForge (http://www.sourceforge.net), Google Code (http://code.google.com) or GitHub (https://github.com/). You should expect to run into bugs that you simply don't understand while working on this and subsequent projects. When you do, reread the appendix on debugging tools, which is filled with useful debugging tips that should help you to get back up to speed. In addition, the YouTube video should get you up to speed using GitHub – this was the most commonly used version control system in the past.

2. Requirements

2.1 Design Document

We recommend that you read the design document template (Design1.txt) before you start working on the project. You must turn in your project with a copy of the design document in the pintos/src/threads folder.

2.2 Alarm Clock

Re-implement 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. Re-implement 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 t_ticks timer ticks. Unless the system is otherwise idle, the thread need not wake up after exactly t_ticks ticks. Just put it on the ready queue after they have waited for the right amount of time. The function 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 other 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, re-read the explanation of the "-r" option to pintos. The alarm clock implementation is not needed for later projects, but it may be useful for some.

2.3 Priority Scheduling

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_* symbolic constants are defined in "threads/thread.h", and you should not change their values.

One issue with priority scheduling is the problem of "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. In this case, L "inherits" the priority of H while holding the lock.

Implement priority donation (which is also called priority inheritance). 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; that is, priority inheritance is transitive. If necessary, you may impose a reasonable limit on the 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, then it yields.

Function: **`int thread_get_priority (void)`**
Returns the current thread's priority. In the presence of priority donation, returns the highest (donated) priority. You need not provide any interface to allow a thread to directly modify other threads' priorities. The priority scheduler is not used in any later project.

2.4 Advanced MLFQ Scheduler [Extra Credit +10]

Implement a multilevel feedback queue scheduler similar to the 4.4BSD 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. 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 "-mlfqs" 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 function thread_set_priority(), and the function thread_get_priority() should return the thread's current priority as set by the scheduler.
The advanced scheduler is not used in any later project.

Grading

This assignment will count for 50 points (note the first three parts are required):
+20 points: Design document
+15 points: Passing alarm clock test
+15 points: Passing priority scheduling test
+10 points: Passing advanced scheduling test [Extra Credit]