Ardalan Amiri Sani

CompSci 143A: Principles of Operating System

Instructor: Ardalan Amiri Sani

Pintos Project Lab 1: Threads

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 pr [illegible] ent, with some work in the “devices [illegible] e in the “threads” directory.

Before you read the d [illegible] e following sections: 1. Introducti [illegible] d F. Development Tools. You should at le [illegible] .5 Memory Allocation, especially [illegible] will also need to read B. 4.4BSD Sche [illegible]

Background

Understanding Th [illegible]

The first step is to rea [illegible] stem. 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 debug … what happens (see section E.5 GDB). You … and then single-step from there.(1) Be … state, and what procedures are on the … en one thread calls `switch_threads` … ng the new thread does is to return from … d system once you understand why a … different from the `switch_threads()` th… r more information.

Warning!

In Pintos, each thread … st under 4 kB in size. The kernel tries t… ctly. You may cause bizarre problem… e large data structures as non-stat… natives to stack allocation include the page allocator and the block allocator (see section A.5 Memory Allocation).

Source Files

Refer to link for a brief overview of the files in the “`threads`” directory. You will not need to modify most of this code, but the hope is that presenting this overview will give you a start on what code to look at.

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.

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 in [illegible] ad variables. When you access the [illegible] disable interrupts to prevent the timer in [illegible]

When you do turn off [illegible] of code possible, or you can end up los [illegible] vents. Turning off interrupts also increas [illegible] a machine feel sluggish if taken too fa [illegible]

The synchronization p [illegible] ed by disabling interrupts. You may ne [illegible] interrupts disabled here, but you [illegible]

Disabling interrupts ca [illegible] re that a section of code is not interrup [illegible] rning in your project. (Don't just co [illegible] icult to read.)

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

Development Suggestions

In the past, many groups divided the assignment into pieces, then each group member worked on his or her piece until just before the deadline, at which time the group reconvened to combine their code and submit. **This is a bad idea. We do not recommend this approach.** Groups that do this 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 team's changes early and often, using a source code control system such as Git (see section F.3 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 should expect to run into bugs that you simply don't understand while working on this project. 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 (see section E. Debugging Tools). Be sure to read the section on backtraces (see section E.4 Backtraces), which will help you to get the most out of every kernel panic or assertion failure.

Requirements

---

0. Design Document

Before you turn in you … ument template into your source tree … NDOC" and fill it in. We recommend that y … u start working on the project. See sectio … document that goes along with a ficti …

---

1. Alarm Clock

Exercise 1.1

Reimplement `timer_s` … gh a working implementation is pro … cking the current time and calling `threa` … lement 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.

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).

The alarm clock implementation is not needed for later projects, although it could be useful for project 4.

---

2. Priority Scheduling

Exercise 1.2.1

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 ... ng for a lock, semaphore, or conditi... uld be awakened first. A thread may rai... g its priority such that it no longer has th... eld the CPU.

Thread priorities range... rs correspond to lower priorities, so tha... he highest. The initial thread priority is... here's no reason to choose another pri... defined in "`threads/thread.h`...

One issue with priority... medium, and low priority threads *H*, *M*, ... stance, for a lock held by *L*), and *M* is o... ause the low priority thread will not ... for *H* to "donate" its priority to *L* while *L* ... *L* releases (and thus *H* acquires) the lock.

Exercise 1.2.2

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.

Exercise 1.2.3

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.

The priority scheduler [illegible]

---

3. Advanced Sche[illegible]

Exercise 1.3

Implement a multileve[illegible] scheduler to reduce the average re[illegible] section B. 4.4BSD Scheduler, fo[illegible]
Like the priority sched[illegible] to run based on priorities. However, th[illegible]n. Thus, we recommend that you [illegible]ly for priority donation, before you [illegible]

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 `thread_set_priority()`, and `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.
