CS 162 Project 1: `Threads`

September 12, 2014

Contents

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.

1 Pintos

1.1 What is Pintos?

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. In the Pintos projects, you and your project team will strengthen its support in all three of these areas. You will also add a virtual memory implementation. Pintos could, theoretically, run on a regular IBM-compatible PC. Unfortunately, it is impractical to supply every CS 162 student a dedicated PC for use with Pintos. Therefore, we will run Pintos projects in 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. In class we will use the Bochs and QEMU simulators. Pintos has also been tested with VMware Player.

1.2 Getting Started

To get started you will have to log in to the vagrant box you set up in HW0, after you clone and build the project, Pintos should just work. All necessary scripts & dependencies to run Pintos have already been installed in the virtual machine.

As this is your first group project, you will need to link your virtual machine to your group's repository. You will need to go into the `group` directory located in the home directory. You will then need to build Pintos in order to use it.

Replace groupX with the group repo you were assigned to!

```
cd ~/group
git remote rm staff
git remote rm group
git remote add staff git@github.com:Berkeley-CS162/group0.git
git pull staff master
git remote add group git@github.com:Berkeley-CS162/groupX.git
git push group master
git checkout -b ag/project1
git checkout -b release/project1
git checkout master
cd src/utils/
make
cd ../threads
make check
```

The last command should run our (non-comprehensive) test suite and should say you've failed all the tests. Your job is to fix that.

At this point you should have a nice, clean, populated git repo that contains the sources for your group to work on. You may want to tell git to push and pull from your group repository by default (as opposed to origin, which doesn't exist). You can do this by running the following commands:

```
$ git branch --set-upstream master group/master
```

1.2.1 Installing on own computer

Some of you Linux users may want to forgo the entire virtual machine altogether and want to work natively on your machine. Though the course doesn't "officially" support this you can find instructions for doing so here

1.3 Source Tree

`threads/`
Source code for the base Pintos kernel, most of your Project 1 work will reside here

`userprog/`
Source code for the user program loader, you will modify this for Project 2

`vm/`
An almost empty directory. You will implement virtual memory here in Project 2.

`filesys/`
Source code for a basic file system. You will use this file system in project 2.

`devices/`
Source code for I/O device interfacing: keyboard, timer, disk, etc. You will modify the timer implementation in project 1. Otherwise you should have no need to change this code.

`lib/`
An implementation of a subset of the standard C library. The code in this directory is compiled into both the Pintos kernel and, starting from project 2, user programs that run under it. In both kernel code and user programs, headers in this directory can be included using the #include <...> notation. You should have little need to modify this code.

`lib/kernel/`
Parts of the C library that are included only in the Pintos kernel. This also includes implementations of some data types that you are free to use in your kernel code: bitmaps, doubly linked lists, and hash tables. In the kernel, headers in this directory can be included using the include <...> notation.

`lib/user/`
Parts of the C library that are included only in Pintos user programs. In user programs, headers in this directory can be included using the #include <...> notation.

`tests/`
Tests for each project. You can modify this code if it helps you test your submission, but we will replace it with the originals before we run the tests.

`examples/`
Example user programs for use starting with project 2.

`misc/`
`utils/`
These files may come in handy if you decide to try working with Pintos on your own machine. Otherwise, you can ignore them.

1.4 Building Pintos

This section describes the interesting files inside the `\src\threads\build` directory.

`threads/build/Makefile`
A copy of pintosMakefile.build. It describes how to build the kernel. See Adding Source Files, for more information.

`threads/build/kernel.o`
Object file for the entire kernel. This is the result of linking object files compiled from each individual kernel source file into a single object file. It contains debug information, so you can run GDB or back-trace on it.

`threads/build/kernel.bin`
Memory image of the kernel, that is, the exact bytes loaded into memory to run the Pintos kernel. This is just kernel.o with debug information stripped out, which saves a lot of space, which in turn keeps the kernel from bumping up against a 512 kB size limit imposed by the kernel loader's design.

`threads/build/loader.bin`
Memory image for the kernel loader, a small chunk of code written in assembly language that reads the kernel from disk into memory and starts it up. It is exactly 512 bytes long, a size fixed by the PC BIOS.

Subdirectories of build contain object files (.o) and dependency files (.d), both produced by the compiler. The dependency files tell make which source files need to be recompiled when other source or header files are changed.

1.5 Running Pintos

Note: If you are not using the cs162 Vagrant box, you can run pintos by logging into icluster22.eecs.berkeley.edu, icluster23, or icluster24. Pintos will only work on these machines. Please let a TA know via Piazza if Pintos is not working on these machines.

We've supplied a program for conveniently running Pintos in a simulator, called `pintos`. In the simplest case, you can invoke pintos as `pintos argument...`. Each argument is passed to the Pintos kernel for it to act on.

Try it out. First `cd` into the newly created build directory. Then issue the command `pintos run alarm-multiple`, which passes the arguments `run alarm-multiple` to the Pintos kernel. In these arguments, run instructs the kernel to run a test and alarm-multiple is the test to run.

This command creates a bochsrc.txt file, which is needed for running Bochs, and then invoke Bochs. The text printed by Pintos inside Bochs probably went by too quickly to read. However, you've probably noticed by now that the same text was displayed in the terminal you used to run pintos. This is because Pintos sends all output both to the VGA display and to the first serial port, and by default the serial port is connected to Bochs's stdin and stdout. You can log serial output to a file by redirecting at the command line, e.g. `pintos run alarm-multiple > logfile`.

The `pintos` program offers several options for configuring the simulator or the virtual hardware. If you specify any options, they must precede the commands passed to the Pintos kernel and be separated from them by –, so that the whole command looks like `pintos option... -- argument...` Invoke pintos without any arguments to see a list of available options. Options can select a simulator to use: the default is Bochs, but `--qemu` selects QEMU. You can set the amount of memory to give the VM. You can run the simulator with a debugger. GDB must be attached to pintos via the following command:

```
(gdb) target remote localhost:1234
```

Finally, you can select how you want VM output to be displayed: use `-v` to turn off the VGA display, `-t` to use your terminal window as the VGA display instead of opening a new window (Bochs only), or `-s` to suppress serial input from stdin and output to stdout.

The Pintos kernel has commands and options other than run. These are not very interesting for now, but you can see a list of them using `-h`, e.g. `pintos -h`.

1.6 More

Pintos is a small operating system, but it is still a valid operating system, so there is a lot of complexity, lot more than I can fit in one section of a Project spec, fortunately we had a detailed reference guide availble on the course website here

1.7 Why Pintos?

Why the name "Pintos"? First, like nachos (the operating system previously used in CS162), pinto beans are a common Mexican food. Second, Pintos is small and a "pint" is a small amount. Third, like drivers of the eponymous car, students are likely to have trouble with blow-ups.

2 Threads

Alright enough background information, lets get started with our project!

2.1 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. 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. You can set a breakpoint on `schedule()` to start out, and then single-step from there.(2) 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.

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).

2.2 Source Files

You will want to look at all the files inside `threads/` and `devices/` you will most certainly not change all the files in here but its good practice to look through all the code to see what kind of beast you are working with. It may also help to look through code in the `lib/` directory to understand the useful library routines Pintos uses.

2.3 Synchronization

Proper synchronization is an important part of the solutions to these problems. Any synchronization problem can be crudely 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 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.

2.4 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.

- `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. The alarm clock implementation is not needed for later projects.

2.5 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.

2.5.1 Priority Donation

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, fix the following functions that allow a thread to examine and modify its own priority. Skeletons for these functions are provided in threads/thread.c.

- `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.

- `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 is not used in any later project.

3 ε Bonus

ε Bonus doesn't have any points.

3.1 Advanced 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 here, 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 `-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

4 Testing

Your test result 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. For project 1, the tests will probably run faster in Bochs. For the rest of the projects, they will run much faster in QEMU. make check will select the faster simulator by default, but you can override its choice by specifying `SIMULATOR=--bochs` or `SIMULATOR=--qemu` on the make command line.

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/

All of the tests and related files are in pintos/src/tests. Before we test your submission, we will replace the contents of that directory by a pristine, unmodified copy, to ensure that the correct tests are used. Thus, you can modify some of the tests if that helps in debugging, but we will run the originals.

All software has bugs, so some of our tests may be flawed. If you think a test failure is a bug in the test, not a bug in your code, please point it out. We will look at it and fix it if necessary.

Please don't try to take advantage of our generosity in giving out our test suite. Your code has to work properly in the general case, not just for the test cases we supply. For example, it would be unacceptable to explicitly base the kernel's behavior on the name of the running test case. Such attempts to side-step the test cases will receive no credit. If you think your solution may be in a gray area here, please ask us about it.

5 Schedule and Grading

As this assignment is fairly open-ended, we'll have weekly check points to ensure that you're on the right track. The assignment lasts for three weeks. The first two deadlines are small bite sized portions of your project for your benefit: feel free to submit early if you're looking for feedback. Missing these deadlines causes you to lose 5 points (each) on your code portion. You **may not** use slip days for checkpoints 1 or 2. Whereas the final deadline is worth 50 points and you may use slip days. All checkpoints are due on Wednesday nights.

Additionally, the design you provide during your first two weeks is considered to be a work-in-progress. I'm expecting that you'll make some changes to your design during implementation and that you'll need to change your tests to accommodate interesting cases you discover while trying to make things work. That said, you'll be penalized for any major design changes. In other words: your design document needs to be a good-faith effort and reasonably resemble what you build, otherwise you haven't really done a design document. We're trying to get you into the habit of test-driven development, which means your tests should be good enough to describe how your system should work before you start implementation. To enforce this, a similar penalty policy exists for your tests: if your final set of tests don't look anything like your earlier tests, then we'll take off points as you haven't really built proper tests the first time. That said, I wouldn't worry about it – nobody lost points for this last semester.

If you miss a checkpoint deadline then you can submit working code up until the final deadline for credit (you just lose the points associated with that checkpoint). Note that all the checkpoints require you to submit partial functionality so you'll probably have to end up doing the work anyway in order to do the next checkpoint.

5.1 Checkpoint 1

Due: 9/24
For the first checkpoint, the following code features will be due

- (5 points) A completely working Alarm Clock implementation that passes all our given tests.
- (10 points) An initial design document that details how you will implement priority scheduler and priority donation and how you have implemented alarm.
- (10 points) Your design review with your TA (being able to answer questions about the project)

5.2 Checkpoint 2

Due: 10/01

- (5 points) A Priority Scheduler that passes all our tests EXCEPT donation

5.3 Final Code Handin

Due: 10/08

- (50 points) A fully functional Priority Scheduler with donation and a fully functional alarm.
- (10 points) Good design and clean code through out your entire project

5.4 Final Report Handin

Due: 10/10

- (10 points) Final Report (A cleaned up version of your initial design document)

6 Advice

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 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 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. Be sure to read the section on backtraces in the appendix, which will help you to get the most out of every kernel panic or assertion failure.

We also encourage you guys to pair or even group program. Having multiple sets of eyes looking at the same code can help avoid/spot subtle bugs that can drive people insane.

Also use gdb.

Do not commit/push binary files.

These projects are designed to be difficult and even push you to your limits as a developer, so plan to be busy the next three weeks, and have fun!

A 4.4BSD Scheduler

The goal of a general-purpose scheduler is to balance threads' different scheduling needs. Threads that perform a lot of I/O require a fast response time to keep input and output devices busy, but need little CPU time. On the other hand, compute-bound threads need to receive a lot of CPU time to finish their work, but have no requirement for fast response time. Other threads lie somewhere in between, with periods of I/O punctuated by periods of computation, and thus have requirements that vary over time. A well-designed scheduler can often accommodate threads with all these requirements simultaneously.

For project 1, you must implement the scheduler described in this appendix. Our scheduler resembles the one described in [McKusick], which is one example of a *multilevel feedback queue* scheduler. 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 non-empty queue. If the highest-priority queue contains multiple threads, then they run in "round robin" order.

Multiple facets of the scheduler require data to be updated after a certain number of timer ticks. In every case, these updates should occur before any ordinary kernel thread has a chance to run, so that there is no chance that a kernel thread could see a newly increased `timer_ticks()` value but old scheduler data values.

The 4.4BSD scheduler does not include priority donation.

A.1 Niceness

Thread priority is dynamically determined by the scheduler using a formula given below. However, each thread also has an integer `nice` value that determines how "nice" the thread should be to other threads. A `nice` of zero does not affect thread priority. A positive `nice`, to the maximum of 20, decreases the priority of a thread and causes it to give up some CPU time it would otherwise receive. On the other hand, a negative `nice`, to the minimum of -20, tends to take away CPU time from other threads. The initial thread starts with a `nice` value of zero. Other threads start with a `nice` value inherited from their parent thread. You must implement the functions described below, which are for use by test programs. We have provided skeleton definitions for them in "`threads/thread.c`".

- `int thread_get_nice (void)`

  Returns the current thread's `nice` value.

- `void thread_set_nice (int new_nice)`

  Sets the current thread's `nice` value to *new_nice* and recalculates the thread's priority based on the new value (see section A.2 Calculating Priority). If the running thread no longer has the highest priority, yields.

A.2 Calculating Priority

Our scheduler has 64 priorities and thus 64 ready queues, numbered 0 (`PRI_MIN`) through 63 (`PRI_MAX`). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. Thread priority is calculated initially at thread initialization. It is also recalculated once every fourth clock tick, for every thread. In either case, it is determined by the formula

$$\texttt{priority} = \texttt{PRI\_MAX} - (\texttt{recent\_cpu}/4) - (\texttt{nice} * 2)$$

, where `recent_cpu` is an estimate of the CPU time the thread has used recently (see below) and `nice` is the thread's `nice` value. The result should be rounded down to the nearest integer (truncated). The coefficients 1/4 and 2 on `recent_cpu` and `nice`, respectively, have been found to work well in practice but lack deeper meaning. The calculated `priority` is always adjusted to lie in the valid range `PRI_MIN` to `PRI_MAX`.

This formula gives a thread that has received CPU time recently lower priority for being reassigned the CPU the next time the scheduler runs. This is key to preventing starvation: a thread that has not received any CPU time recently will have a `recent_cpu` of 0, which barring a high `nice` value should ensure that it receives CPU time soon.

A.3 Calculating recent_cpu

We wish `recent_cpu` to measure how much CPU time each process has received "recently." Furthermore, as a refinement, more recent CPU time should be weighted more heavily than less recent CPU time. One approach would use an array of n elements to track the CPU time received in each of the last n seconds. However, this approach requires O(n) space per thread and O(n) time per calculation of a new weighted average.

Instead, we use a *exponentially weighted moving average*, which takes this general form:

$$x(0) = f(0)$$
$$x(t) = a * x(t-1) + f(t)$$
$$a = k/(k+1)$$

where x(t) is the moving average at integer time t >= 0, f(t) is the function being averaged, and k >0 controls the rate of decay. We can iterate the formula over a few steps as follows:

$$x(1) = f(1)$$
$$x(2) = a * f(1) + f(2)$$
$$...$$
$$x(5) = a^4 * f(1) + a^3 * f(2) + a^2 * f(3) + a * f(4) + f(5)$$

The value of f(t) has a weight of 1 at time t, a weight of a at time t+1, a^2 at time t+2, and so on. We can also relate x(t) to k: f(t) has a weight of approximately 1/e at time t+k, approximately $1/e^2$ at time t+2*k, and so on. From the opposite direction, f(t) decays to weight w at time $t + ln(w)/ln(a)$.

The initial value of `recent_cpu` is 0 in the first thread created, or the parent's value in other new threads. Each time a timer interrupt occurs, `recent_cpu` is incremented by 1 for the running thread only, unless the idle thread is running. In addition, once per second the value of `recent_cpu` is recalculated for every thread (whether running, ready, or blocked), using this formula:

$$\texttt{recent\_cpu} = (2 * \texttt{load\_avg})/(2 * \texttt{load\_avg} + 1) * \texttt{recent\_cpu} + \texttt{nice}$$

, where `load_avg` is a moving average of the number of threads ready to run (see below). If `load_avg` is 1, indicating that a single thread, on average, is competing for the CPU, then the current value of `recent_cpu` decays to a weight of 0.1 in $\ln(0.1)/\ln(\frac{2}{3})$ = approx. 6 seconds; if `load_avg` is 2, then decay to a weight of 0.1 takes $ln(0.1)/ln(\frac{3}{4})$ = approx. 8 seconds. The effect is that `recent_cpu` estimates the amount of CPU time the thread has received "recently," with the rate of decay inversely proportional to the number of threads competing for the CPU.

Assumptions made by some of the tests require that these recalculations of `recent_cpu` be made exactly when the system tick counter reaches a multiple of a second, that is, when `timer_ticks() % TIMER_FREQ == 0`, and not at any other time.

The value of `recent_cpu` can be negative for a thread with a negative nice value. Do not clamp negative `recent_cpu` to 0.

You may need to think about the order of calculations in this formula. We recommend computing the coefficient of `recent_cpu` first, then multiplying. Some students have reported that multiplying `load_avg` by `recent_cpu` directly can cause overflow.

You must implement `thread_get_recent_cpu()`, for which there is a skeleton in "`threads/thread.c`".

- `int thread_get_recent_cpu(void)`

  Returns 100 times the current thread's `recent_cpu` value, rounded to the nearest integer.

A.4 Calculating load_avg

Finally, `load_avg`, often known as the system load average, estimates the average number of threads ready to run over the past minute. Like `recent_cpu`, it is an exponentially weighted moving average. Unlike priority and `recent_cpu`, `load_avg` is system-wide, not thread-specific. At system boot, it is initialized to 0. Once per second thereafter, it is updated according to the following formula:

$$\texttt{load\_avg} = (59/60) * \texttt{load\_avg} + (1/60) * \texttt{ready\_threads}$$

, where `ready_threads` is the number of threads that are either running or ready to run at time of update (not including the idle thread).

Because of assumptions made by some of the tests, `load_avg` must be updated exactly when the system tick counter reaches a multiple of a second, that is, when `timer_ticks() % TIMER_FREQ == 0`, and not at any other time.

You must implement `thread_get_load_avg()`, for which there is a skeleton in "`threads/thread.c`".

- `int thread_get_load_avg(void)`

  Returns 100 times the current system load average, rounded to the nearest integer.

A.5 Summary

The following formulas summarize the calculations required to implement the scheduler. They are not a complete description of scheduler requirements.

Every thread has a nice value between -20 and 20 directly under its control. Each thread also has a priority, between 0 (`PRI_MIN`) through 63 (`PRI_MAX`), which is recalculated using the following formula every fourth tick:

$$\texttt{priority} = \texttt{PRI\_MAX} - (\texttt{recent\_cpu}/4) - (\texttt{nice} * 2)$$

. `recent_cpu` measures the amount of CPU time a thread has received "recently." On each timer tick, the running thread's `recent_cpu` is incremented by 1. Once per second, every thread's `recent_cpu` is updated this way:

$$\texttt{recent\_cpu} = (2 * \texttt{load\_avg})/(2 * \texttt{load\_avg} + 1) * \texttt{recent\_cpu} + \texttt{nice}$$

. `load_avg` estimates the average number of threads ready to run over the past minute. It is initialized to 0 at boot and recalculated once per second as follows:

$$\texttt{load\_avg} = (59/60) * \texttt{load\_avg} + (1/60) * \texttt{ready\_threads}$$

. where `ready_threads` is the number of threads that are either running or ready to run at time of update (not including the idle thread).

A.6 Fixed-Point Real Arithmetic

In the formulas above, `priority`, `nice`, and `ready_threads` are integers, but `recent_cpu` and `load_avg` are real numbers. Unfortunately, Pintos does not support floating-point arithmetic in the kernel, because it would complicate and slow the kernel. Real kernels often have the same limitation, for the same reason. This means that calculations on real quantities must be simulated using integers. This is not difficult, but many students do not know how to do it. This section explains the basics.

The fundamental idea is to treat the rightmost bits of an integer as representing a fraction. For example, we can designate the lowest 14 bits of a signed 32-bit integer as fractional bits, so that an integer x represents the real number $x/(2^{14})$.This is called a 17.14 fixed-point number representation, because there are 17 bits before the decimal point, 14 bits after it, and one sign bit. [1] A number in 17.14 format represents, at maximum, a value of $(2^{31} - 1)/(2^{14})$ = approx. 131,071.999.

Suppose that we are using a p.q fixed-point format, and let f = 2**q. By the definition above, we can convert an integer or real number into p.q format by multiplying with f. For example, in 17.14 format the fraction 59/60 used in the calculation of `load_avg`, above, is $59/60 * (2 * *14) = 16,110$. To convert a fixed-point value back to an integer, divide by f. (The normal "/" operator in C rounds toward zero, that is, it rounds positive numbers down and negative numbers up. To round to nearest, add f / 2 to a positive number, or subtract it from a negative number, before dividing.)

Many operations on fixed-point numbers are straightforward. Let x and y be fixed-point numbers, and let n be an integer. Then the sum of x and y is $x + y$ and their difference is $x - y$. The sum of x and n is $x + n * f$; difference, $x - n * f$; product, $x * n$; quotient, x/n.

Multiplying two fixed-point values has two complications. First, the decimal point of the result is q bits too far to the left. Consider that $\frac{59}{60} * \frac{59}{60}$ should be slightly less than 1, but $16,111 * 16,111 = 259,564,321$ is much greater than $2^{14} = 16,384$. Shifting q bits right, we get $259,564,321/(2^{14}) = 15,842$, or about 0.97, the correct answer. Second, the multiplication can overflow even though the answer is representable. For example, 64 in 17.14 format is $64 * (2 * *14) = 1,048,576$ and its square $64^2 = 4,096$ is well within the 17.14 range, but $1,048,576^2 = 2^{40}$, greater than the maximum signed 32-bit integer value 2**31 - 1. An easy solution is to do the multiplication as a 64-bit operation. The product of x and y is then $((int64_t)x) * y/f$.

Dividing two fixed-point values has opposite issues. The decimal point will be too far to the right, which we fix by shifting the dividend q bits to the left before the division. The left shift discards the top q bits of the dividend, which we can again fix by doing the division in 64 bits. Thus, the quotient when x is divided by y is $((int64_t)x) * f/y$.

This section has consistently used multiplication or division by f, instead of q-bit shifts, for two reasons. First, multiplication and division do not have the surprising operator precedence of the C shift operators. Second, multiplication and division are well-defined on negative operands, but the C shift operators are not. Take care with these issues in your implementation.

The following table summarizes how fixed-point arithmetic operations can be implemented in C. In the table, x and y are fixed-point numbers, n is an integer, fixed-point numbers are in signed p.q format where p + q = 31, and f is $1 << q$:

[1] Because we are working in binary, the "decimal" point might more correctly be called the "binary" point, but the meaning should be clear.

Convert n to fixed point:	$n * f$
Convert x to integer (rounding toward zero):	x/f
Convert x to integer (rounding to nearest):	$(x + f/2)/f$ if $x >= 0$, $(x - f/2)/f$ if $x <= 0$.
Add x and y:	$x + y$
Subtract y from x:	$x - y$
Add x and n:	$x + n * f$
Subtract n from x:	$x - n * f$
Multiply x by y:	$((int64_t)x) * y/f$
Multiply x by n:	$x * n$
Divide x by y:	$((int64_t)x) * f/y$
Divide x by n:	x/n