CS 162 Project 2: `Scheduling`

Design Document Due:	Friday, October 16, 2020
[Ungraded] Checkpoint Due:	Friday, October 23, 2020
Code Due:	Monday, November 2, 2020
Final Report Due:	Wednesday, November 4, 2020

Contents

1 Your Task

In this project, you will add features to the threading system of the educational operating system Pintos. We will introduce these features briefly and provide more details in the reference material at the end of this document.

In Project 1, each thread that you dealt with (except the init and idle threads) was also a process, with its own address space, data backed by an executable file, and ability to execute in userspace. In this project, we will simplify things by dealing with kernel threads—threads that only execute in the kernel mode and have no userspace component. In particular, the `USERPROG` and `FILESYS` macros will not be defined. You are welcome to build on top of your completed Project 1 code, but this is not required—you may also start fresh from the Pintos skeleton code.

- If you choose to start afresh, make sure to create a branch in Git with your latest work from Project 1. You will build on top of this later on in the class, so it is very important that you do not lose your work from Project 1.
- If you choose to build on your work in Project 1 for this project, keep in mind that the `USERPROG` and `FILESYS` macros are not defined. So, you should try to keep any modifications that you made to `struct thread` or `thread.c` within `#ifdef USERPROG` ... `#endif` blocks, so they do not interfere with this project.

1.1 Task 1: Efficient Alarm Clock

In Pintos, threads may call this function to put themselves to sleep:

```
/**
 * This function 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 number of ticks. 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
 * constant defined in devices/timer.h (spoiler: it's 100 ticks per second).
 */
void timer_sleep (int64_t ticks);
```

`timer_sleep()` is useful for threads that operate in real-time (e.g. for blinking the cursor once per second). The current implementation of `timer_sleep()` is inefficient, because it calls `thread_yield()` in a loop until enough time has passed. This consumes CPU cycles while the thread is waiting. Your task is to re-implement `timer_sleep()` so that it executes efficiently without any "busy waiting".

1.2 Task 2: Priority Scheduler

In Pintos, each thread has a priority value ranging from 0 (`PRI_MIN`) to 63 (`PRI_MAX`). However, the current scheduler does not respect these priority values. You must modify the scheduler so that higher-priority threads always run before lower-priority threads (i.e., strict priority scheduling).

You must also modify the 3 Pintos synchronization primitives (lock, semaphore, condition variable), so that these shared resources prefer higher-priority threads over lower-priority threads.

Additionally, you must implement **priority donation** for Pintos locks. When a high-priority thread (A) has to wait to acquire a lock, which is already held by a lower-priority thread (B), we temporarily raise B's priority to A's priority. A scheduler that does not donate priorities is prone to the problem of **priority inversion** whereby a medium-priority thread runs while a high-priority thread (A) waits on a resource held by a low-priority thread (B). A scheduler that supports priority donation would allow B to run first, so that A, which has the highest priority, can be unblocked. Your implementation of priority donation must handle 1) donations from multiple sources, 2) undoing donations when a lock is released, and 3) nested/recursive donation.

A thread may set its own priority by calling `thread_set_priority(int new_priority)` and get its own priority by calling `thread_get_priority()`.

If a thread no longer has the highest "effective priority" (it called `thread_set_priority()` with a low value or it released a lock), it must immediately yield the CPU to the highest-priority thread.

1.3 Optional Stretch Task: Multi-Level Feedback Queue Scheduler (MLFQS)

In addition to the priority scheduler algorithm, you can, as an **optional stretch task**, implement a multi-level feedback queue scheduler algorithm, which is explained in detail in the reference material. The scheduler will either use the priority scheduling algorithm or the MLFQS algorithm, depending on the value of the variable "`bool thread_mlfqs`" (found in `thread.c`). This variable is toggled with `--mlfqs` command-line option to Pintos.

This has been a mandatory part of the project in past semesters, but **this semester, it is an optional stretch task** for groups that found the project not too challenging. The process for submitting this is the same as for the regular project. We will track who attempts or completes the stretch task, but it is ultimately optional.

2 Deliverables

Your project grade will be made up of 4 components:

- 15% Design Document and Design Review
- 75% Code
- 10% Final Report and Code Quality

2.1 Design Document (Due 10/16) and Design Review

Before you start writing any code for your project, you should create an implementation plan for each feature and convince yourself that your design is correct. For this project, you must **submit a design document** and **attend a design review** with your project TA.

2.1.1 Design Document Guidelines

Submit your design document as a PDF to the Project 2 Design Document assignment on Gradescope.

For each of the 2 mandatory tasks of this project, you must explain the following 4 aspects of your proposed design. We suggest you create a section for each of the 2 project tasks. Then, create subsections for each of these 4 aspects.

1. **Data structures and functions** – Write down any struct definitions, global (or static) variables, typedefs, or enumerations that you will be adding or modifying (if it already exists). These definitions should be written with the **C programming language**, not with pseudocode. Include a **brief explanation** the purpose of each modification. Your explanations should be as concise as possible. Leave the full explanation to the following sections.

2. **Algorithms** – This is where you tell us how your code will work. Your description should be at a level below the high level description of requirements given in the assignment. We have read the project spec too, so it is unnecessary to repeat or rephrase what is stated here. On the other hand, your description should be at a level above the code itself. Don't give a line-by-line run-down of what code you plan to write. Instead, you should try to convince us that your design satisfies all the requirements, **including any uncommon edge cases**.

   The length of this section depends on the complexity of the task and the complexity of your design. Simple explanations are preferred, but if your explanation is vague or does not provide enough details, you will be penalized. Here are some tips:

   - For complex tasks, like the priority scheduler, we recommend that you split the task into parts. Describe your algorithm for each part in a separate section. Start with the simplest component and build up your design, one piece at a time. For example, your algorithms section for the Priority Scheduler could have sections for:
     - Choosing the next thread to run
     - Acquiring a Lock
     - Releasing a Lock
     - Computing the effective priority
     - Priority scheduling for semaphores and locks
     - Priority scheduling for condition variables
     - Changing thread's priority
   - Lists can make your explanation more readable. If your paragraphs seem to lack coherency, consider using a list.

- A good length for this section could be 1 paragraph for a simple task (Alarm Clock) or 2 screen pages for a complex task (Priority Scheduler). Make sure your explanation covers all of the required features.
- We fully expect you to read a lot of Pintos code to prepare for the design document. You won't be able to write a good description of your algorithms if you don't know any specifics about Pintos.

3. **Synchronization** – Describe your strategy for preventing race conditions and convince us that it works in all cases. Here are some tips for writing this section:
   - This section should be structured as a **list of all potential concurrent accesses to shared resources**. For each case, you should prove that your synchronization design ensures correct behavior.
   - An operating system kernel is a complex, multithreaded program, in which synchronizing multiple threads can be difficult. The best synchronization strategies are simple and easily verifiable, which leaves little room for mistakes. If your synchronization strategy is difficult to explain, consider how you could simplify it.
   - You should also aim to make your synchronization as efficient as possible, in terms of time and memory.
   - Synchronization issues revolve around shared data. A good strategy for reasoning about synchronization is to identify which pieces of data are accessed by multiple independent actors (whether they are threads or interrupt handlers). Then, prove that the shared data always remains consistent.
   - Lists are a common cause of synchronization issues. Lists in Pintos are not thread-safe.
   - Do not forget to consider memory deallocation as a synchronization issue. If you want to use pointers to `struct thread`, then you need to prove those threads can't exit and be deallocated while you're using them.
   - If you create new functions, you should consider whether the function could be called in 2 threads at the same time. If your function access any global or static variables, you need to show that there are no synchronization issues.
   - Interrupt handlers cannot acquire locks. If you need to access a synchronized variable from an interrupt handler, consider disabling interrupts.
   - Locks do not prevent a thread from being preempted. Threads can be interrupted during a critical section. Locks only guarantee that the critical section is only entered by one thread at a time.
4. **Rationale** – Tell us why your design is better than the alternatives that you considered, or point out any shortcomings it may have. You should think about whether your design is easy to conceptualize, how much coding it will require, the time/space complexity of your algorithms, and how easy/difficult it would be to extend your design to accommodate additional features.

2.1.2 Design Document Additional Questions

You must also answer these additional questions in your design document:

1. In class, we studied the three most important attributes of a thread that the operating system stores when the thread is not running: program counter, stack pointer, and registers. Where/how are each of these three attributes stored in Pintos? You may find it useful to closely read switch.S[1]

[1] https://github.com/Berkeley-CS162/group0/blob/master/pintos/src/threads/switch.S

and the `schedule` function in `thread.c`. You may also find it useful to look over the slides from Lecture 7[2] and Lecture 8[3] of Summer 2020's CS162 offering.

2. When a kernel thread in Pintos calls `thread_exit`, when/where is the page containing its stack and TCB (i.e., `struct thread`) freed? Why can't we just free this memory by calling `palloc_free_page` inside the `thread_exit` function?

3. When the `thread_tick` function is called by the timer interrupt handler, in which stack does it execute?

4. Consider a fully-functional correct implementation of this project, except for a single bug, which exists in the `sema_up()` function. According to the project requirements, semaphores (and other synchronization variables) must prefer higher-priority threads over lower-priority threads. However, the implementation chooses the highest-priority thread based on the **base priority** rather than the **effective priority**. Essentially, priority donations are **not taken into account** when the semaphore decides which thread to unblock. **Please design a test case that can prove the existence of this bug.** Pintos test cases contain regular kernel-level code (variables, function calls, if statements, etc) and can print out text. We can compare the expected output with the actual output. If they do not match, then it proves that the implementation contains a bug. **You should provide a description of how the test works, as well as the expected output and the actual output.**

2.1.3 Design Review

You will schedule a 30 minute design review with your project TA. During the design review, your TA will ask you questions about your design for the project. You should be prepared to defend your design and answer any clarifying questions your TA may have about your design document. The design review is also a good opportunity to get to know your TA for those participation points.

2.1.4 Grading

The design document and design review will be graded together. You will receive a score out of 20 points, which will reflect how convincing your design is, based on your explanation in your design document and your answers during the design review. You **must** attend a design review in order to get these points. We will try to accommodate any time conflicts, but you should let your TA know as soon as possible.

2.2 Code (Due 11/02)

The code section of your grade will be determined by your autograder score. Pintos comes with a test suite that you can run locally on your VM. We run the same tests on the autograder. The results of these tests will determine your code score.

You can check your current grade for the code portion at any time by logging in to the course autograder. Autograder results will also be emailed to you.

We will check your progress on Project 2 at one intermediate checkpoint. The requirements for this checkpoint are described below. **This checkpoint will not be counted towards the final grade for your project.** However, it is in your best interest to complete them to ensure that your group is on pace to finish the assignment. Our goal is not to grade your in-progress implementations, but to ensure that you're making satisfactory progress and encourage you to ask for help early and often.

[2] https://inst.eecs.berkeley.edu/ cs162/su20/static/lectures/7.pdf
[3] https://inst.eecs.berkeley.edu/ cs162/su20/static/lectures/8.pdf

2.3 Checkpoint [Ungraded] (Due 10/23)

You should have implemented Task 1: Efficient Alarm Clock by the checkpoint deadline. Keep in mind that Task 2: Priority Scheduler is significantly more time-consuming to implement, so you may wish to begin Task 2 by this date even though it is not part of this checkpoint.

2.4 Final Code (Due 11/02)

You must have completed both coding tasks (Task 1: Efficient Alarm Clock and Task 2: Priority Scheduler) in their entirety (**??** will be due with your final report).

2.5 Final Report (Due 11/04) and Code Quality

Submit your final report in PDF form to the Project 2 Final Report assignment on Gradescope. Please include the following in your final report:

- the changes you made since your initial design document and why you made them (feel free to re-iterate what you discussed with your TA in the design review)
- a reflection on the project – what exactly did each member do? What went well, and what could be improved?

You will also be graded on the quality of your code. This will be based on many factors:

- Does your code exhibit any major memory safety problems (especially regarding C strings), memory leaks, poor error handling, or race conditions?
- Is your code simple and easy to understand?
- If you have very complex sections of code in your solution, did you add enough comments to explain them?
- Did you leave commented-out code in your final submission?
- Did you copy-paste code instead of creating reusable functions?
- Are your lines of source code excessively long? (more than 100 characters)
- Is your Git commit history full of binary files? (don't commit object files or log files, unless you actually intend to)

3 Reference

3.1 Pintos

In this project, you will be working with *kernel threads*, that is, threads that operate in the kernel without any userspace component. This will allow us to focus on scheduling in this project without worrying about userspace-level concerns.

3.1.1 Getting Started

Log in to the Vagrant Virtual Machine that you set up in Homework 0. You should already have your Pintos code from Project 1 in `~/code/group` on your VM. You may start Project 2 using your Project 1 code. But you may also start over from the skeleton code if you wish.

If you do decide to start over from the skeleton code, please **do not force push** and please do not delete your commits from Project 1. You should know that orphan commits are still accessible on GitHub, and we have a history of the commit hashes you've pushed to the autograder, but we will not enjoy digging up that information if we need it.

If you would like to start over from the skeleton code, please run these commands on your VM:

```
$ cd ~/code/group/
$ git checkout staff/master -- pintos/
$ git commit -m "Revert changes to pintos/ from Project 1"
$ git push group master
```

We recommend that you first use Git to tag your final Project 1 code, for your own benefit. You will be building on it later in the course.

Once you have made some progress on your project, you can push your code to the autograder by pushing to "`group master`". This will use the "group" remote that we just set up. You don't have to do this right now, because you haven't made any progress yet.

```
$ git commit -m "Added feature X to Pintos"
$ git push group master
```

To compile Pintos and run the Project 2 tests:

```
$ cd ~/code/group/pintos/src/threads
$ make
$ make check
```

The last command should run the Pintos test suite. These are the same tests that run on the autograder. The skeleton code already passes some of these tests. By the end of the project, your code should pass all of the tests.

3.1.2 Source Tree

In the Project 1 specification, we provided an overview of the Pintos source tree. Here, we focus on the parts that we expect you to modify for Project 2.

`threads/`
The base Pintos kernel. Most of the modifications you will make for Project 2 will be in this directory.

`devices/`
Source code for I/O device interfacing: keyboard, timer, disk, etc. You will modify the timer implementation in Project 2.

`tests/`
Tests for each project. You can add extra tests, but do not modify the given tests.

3.2 Threads

3.2.1 Understanding Threads

The first step is to read and understand the code for the 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, step through code and examine data, and so on.

When a thread is created, the creator specifies a function for the thread to run, as one of the arguments to `thread_create()`. The first time the thread is scheduled and runs, it starts executing from the beginning of that function. 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 become inactive. The scheduler decides which thread to run next. (If no thread is ready to run, then the special "idle" thread runs.)

The mechanics of a context switch are in `threads/switch.S`, which is x86 assembly code. It saves the state of the currently running thread and restores the state of the next thread onto the CPU.

Using GDB, try tracing through a context switch to see what happens. You can set a breakpoint on `schedule()` to start out, and then single-step from there (use "`step`" instead of "`next`"). Be sure to keep track of each thread's address and state, and what procedures are on the call stack for each thread (try "`backtrace`"). 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.

3.2.2 The Thread Struct

Each thread struct represents either a kernel thread or a user process. In each of the 3 projects, you will have to add your own members to the thread struct. You may also need to change or delete the definitions of existing members.

Every thread struct occupies the beginning of its own 4KiB page of memory. The rest of the page is used for the thread's stack, which grows downward from the end of the page. It looks like this:

```
4 kB +---------------------------------+
     |          kernel stack           |
     |                |                |
     |                |                |
     |                V                |
     |         grows downward          |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
     |                                 |
```

```
                          |                                 |
sizeof (struct thread) +---------------------------------+
                          |               magic             |
                          |                 :               |
                          |                 :               |
                          |               status            |
                          |                tid              |
                     0 kB +---------------------------------+
```

This layout has two consequences. First, struct thread must not be allowed to grow too big. If it does, then there will not be enough room for the kernel stack. The base struct thread is only a few bytes in size. It probably should stay well under 1 kB.

Second, kernel stacks must not be allowed to grow too large. If a stack overflows, it will corrupt the thread state. Thus, kernel functions should not allocate large structures or arrays as non-static local variables. Use dynamic allocation with `malloc()` or `palloc_get_page()` instead. See the Memory Allocation section for more details.

- **Member of struct thread:** `tid_t tid`
  The thread's thread identifier or *tid*. Every thread must have a tid that is unique over the entire lifetime of the kernel. By default, `tid_t` is a `typedef` for `int` and each new thread receives the numerically next higher tid, starting from 1 for the initial process.

- **Member of struct thread:** `enum thread_status status`
  The thread's state, one of the following:

  - **Thread State:** `THREAD_RUNNING`
    The thread is running. Exactly one thread is running at a given time. `thread_current()` returns the running thread.
  - **Thread State:** `THREAD_READY`
    The thread is ready to run, but it's not running right now. The thread could be selected to run the next time the scheduler is invoked. Ready threads are kept in a doubly linked list called `ready_list`.
  - **Thread State:** `THREAD_BLOCKED`
    The thread is waiting for something, e.g. a lock to become available, an interrupt to be invoked. The thread won't be scheduled again until it transitions to the `THREAD_READY` state with a call to `thread_unblock()`. This is most conveniently done indirectly, using one of the Pintos synchronization primitives that block and unblock threads automatically.
  - **Thread State:** `THREAD_DYING`
    The thread has exited and will be destroyed by the scheduler after switching to the next thread.

- **Member of struct thread:** `char name[16]`
  The thread's name as a string, or at least the first few characters of it.

- **Member of struct thread:** `uint8_t *stack`
  Every thread has its own stack to keep track of its state. When the thread is running, the CPU's stack pointer register tracks the top of the stack and this member is unused. But when the CPU switches to another thread, this member saves the thread's stack pointer. No other members are needed to save the thread's registers, because the other registers that must be saved are saved on the stack.

  When an interrupt occurs, whether in the kernel or a user program, an "`struct intr_frame`" is pushed onto the stack. When the interrupt occurs in a user program, the "`struct intr_frame`" is always at the very top of the page.

- **Member of struct thread:** `int priority`
  A thread priority, ranging 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. Pintos currently ignores these priorities, but you will implement priority scheduling in this project.

- **Member of struct thread:** `struct list_elem allelem`
  This "list element" is used to link the thread into the list of all threads. Each thread is inserted into this list when it is created and removed when it exits. The `thread_foreach()` function should be used to iterate over all threads.

- **Member of struct thread:** `struct list_elem elem`
  A "list element" used to put the thread into doubly linked lists, either `ready_list` (the list of threads ready to run) or a list of threads waiting on a semaphore in `sema_down()`. It can do double duty because a thread waiting on a semaphore is not ready, and vice versa.

- **Member of struct thread:** `uint32_t *pagedir`
  (Used in Projects 1 and 3.) The page table for the process, if this is a user process.

- **Member of struct thread:** `unsigned magic`
  Always set to `THREAD_MAGIC`, which is just an arbitrary number defined in `threads/thread.c`, and used to detect stack overflow. `thread_current()` checks that the `magic` member of the running thread's `struct thread` is set to `THREAD_MAGIC`. Stack overflow tends to change this value, triggering the assertion. For greatest benefit, as you add members to `struct thread`, leave `magic` at the end.

3.2.3 Thread Functions

`threads/thread.c` implements several public functions for thread support. Let's take a look at the most useful ones:

- **Function:** `void thread_init (void)`
  Called by `main()` to initialize the thread system. Its main purpose is to create a `struct thread` for Pintos's initial thread. This is possible because the Pintos loader puts the initial thread's stack at the top of a page, in the same position as any other Pintos thread.

  Before `thread_init()` runs, `thread_current()` will fail because the running thread's `magic` value is incorrect. Lots of functions call `thread_current()` directly or indirectly, including `lock_acquire()` for locking a lock, so `thread_init()` is called early in Pintos initialization.

- **Function:** `void thread_start (void)`
  Called by `main()` to start the scheduler. Creates the idle thread, that is, the thread that is scheduled when no other thread is ready. Then enables interrupts, which as a side effect enables the scheduler because the scheduler runs on return from the timer interrupt, using `intr_yield_on_return()`.

- **Function:** `void thread_tick (void)`
  Called by the timer interrupt at each timer tick. It keeps track of thread statistics and triggers the scheduler when a time slice expires.

- **Function:** `void thread_print_stats (void)`
  Called during Pintos shutdown to print thread statistics.

- **Function:** `tid_t thread_create (const char *name, int priority, thread_func *func, void *aux)`
  Creates and starts a new thread named `name` with the given `priority`, returning the new thread's tid. The thread executes `func`, passing `aux` as the function's single argument.

`thread_create()` allocates a page for the thread's thread struct and stack and initializes its members, then it sets up a set of fake stack frames for it. The thread is initialized in the blocked state, then unblocked just before returning, which allows the new thread to be scheduled.

  – **Type:** `void thread_func (void *aux)`
    This is the type of the function passed to `thread_create()`, whose `aux` argument is passed along as the function's argument.

- **Function:** `void thread_block (void)`
  Transitions the running thread from the running state to the blocked state. The thread will not run again until `thread_unblock()` is called on it, so you'd better have some way arranged for that to happen. Because `thread_block()` is so low-level, you should prefer to use one of the synchronization primitives instead.

- **Function:** `void thread_unblock (struct thread *thread)`
  Transitions `thread`, which must be in the blocked state, to the ready state, allowing it to resume running. This is called when the event that the thread is waiting for occurs, e.g. when the lock that the thread is waiting on becomes available.

- **Function:** `struct thread *thread_current (void)`
  Returns the running thread.

- **Function:** `tid_t thread_tid (void)`
  Returns the running thread's thread id. Equivalent to `thread_current ()->tid`.

- **Function:** `const char *thread_name (void)`
  Returns the running thread's name. Equivalent to `thread_current ()->name`.

- **Function:** `void thread_exit (void) NO_RETURN`
  Causes the current thread to exit. Never returns, hence `NO_RETURN`.

- **Function:** `void thread_yield (void)`
  Yields the CPU to the scheduler, which picks a new thread to run. The new thread might be the current thread, so you can't depend on this function to keep this thread from running for any particular length of time.

- **Function:** `void thread_foreach (thread_action_func *action, void *aux)`
  Iterates over all threads `t` and invokes `action(t, aux)` on each. `action` must refer to a function that matches the signature given by `thread_action_func()`:

  – **Type:** `void thread_action_func (struct thread *thread, void *aux)`
    Performs some action on a thread, given `aux`.

- **Function:** `int thread_get_priority (void)`
  Function: `void thread_set_priority (int new_priority)`
  Stub to set and get thread priority.

- **Function:** `int thread_get_nice (void)`
  Function: `void thread_set_nice (int new_nice)`
  Function: `int thread_get_recent_cpu (void)`
  Function: `int thread_get_load_avg (void)`
  Stubs for the advanced MLFQS scheduler.

3.2.4 Thread Switching

`schedule()` is responsible for switching threads. It is internal to `threads/thread.c` and called only by the three public thread functions that need to switch threads: `thread_block()`, `thread_exit()`, and `thread_yield()`. Before any of these functions call `schedule()`, they disable interrupts (or ensure that they are already disabled) and then change the running thread's state to something other than running.

`schedule()` is short but tricky. It records the current thread in local variable `cur`, determines the next thread to run as local variable `next` (by calling `next_thread_to_run()`), and then calls `switch_threads()` to do the actual thread switch. The thread we switched to was also running inside `switch_threads()`, as are all the threads not currently running, so the new thread now returns out of `switch_threads()`, returning the previously running thread.

`switch_threads()` is an assembly language routine in `threads/switch.S`. It saves registers on the stack, saves the CPU's current stack pointer in the current `struct thread`'s `stack` member, restores the new thread's `stack` into the CPU's stack pointer, restores registers from the stack, and returns.

The rest of the scheduler is implemented in `thread_schedule_tail()`. It marks the new thread as running. If the thread we just switched from is in the dying state, then it also frees the page that contained the dying thread's `struct thread` and stack. These couldn't be freed prior to the thread switch because the switch needed to use it.

Running a thread for the first time is a special case. When `thread_create()` creates a new thread, it goes through a fair amount of trouble to get it started properly. In particular, the new thread hasn't started running yet, so there's no way for it to be running inside `switch_threads()` as the scheduler expects. To solve the problem, `thread_create()` creates some fake stack frames in the new thread's stack:

- The topmost fake stack frame is for `switch_threads()`, represented by `struct switch_threads_frame`. The important part of this frame is its `eip` member, the return address. We point `eip` to `switch_entry()`, indicating it to be the function that called `switch_entry()`.
- The next fake stack frame is for `switch_entry()`, an assembly language routine in `threads/switch.S` that adjusts the stack pointer, calls `thread_schedule_tail()` (this special case is why `thread_schedule_tail()` is separate from `schedule()`), and returns. We fill in its stack frame so that it returns into `kernel_thread()`, a function in `threads/thread.c`.
- The final stack frame is for `kernel_thread()`, which enables interrupts and calls the thread's function (the function passed to `thread_create()`). If the thread's function returns, it calls `thread_exit()` to terminate the thread.

3.3 Synchronization

If sharing of resources between threads is not handled in a careful, controlled fashion, the result is usually a big mess. This is especially the case in operating system kernels, where faulty sharing can crash the entire machine. Pintos provides several synchronization primitives to help out.

3.3.1 Disabling Interrupts

The crudest way to do synchronization is to disable interrupts, that is, to temporarily prevent the CPU from responding to interrupts. If interrupts are off, no other thread will preempt the running thread, because thread preemption is driven by the timer interrupt. If interrupts are on, as they normally are, then the running thread may be preempted by another at any time, whether between two C statements or even within the execution of one.

Incidentally, this means that Pintos is a "preemptible kernel," that is, kernel threads can be preempted at any time. Traditional Unix systems are "nonpreemptible," that is, kernel threads can only be preempted at points where they explicitly call into the scheduler. (User programs can be preempted at any time in both models.) As you might imagine, preemptible kernels require more explicit synchronization.

You should have little need to set the interrupt state directly. Most of the time you should use the other synchronization primitives described in the following sections. The main reason to disable interrupts is to synchronize kernel threads with external interrupt handlers, which cannot sleep and thus cannot use most other forms of synchronization.

Some external interrupts cannot be postponed, even by disabling interrupts. These interrupts, called **non-maskable interrupts** (NMIs), are supposed to be used only in emergencies, e.g. when the computer is on fire. Pintos does not handle non-maskable interrupts.

Types and functions for disabling and enabling interrupts are in `threads/interrupt.h`.

- **Type:** `enum intr_level`
  One of `INTR_OFF` or `INTR_ON`, denoting that interrupts are disabled or enabled, respectively.
- **Function:** `enum intr_level intr_get_level (void)`
  Returns the current interrupt state.
- **Function:** `enum intr_level intr_set_level (enum intr_level level)`
  Turns interrupts on or off according to `level`. Returns the previous interrupt state.
- **Function:** `enum intr_level intr_enable (void)`
  Turns interrupts on. Returns the previous interrupt state.
- **Function:** `enum intr_level intr_disable (void)`
  Turns interrupts off. Returns the previous interrupt state.

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.

3.3.2 Semaphores

A **semaphore** is a nonnegative integer together with two operators that manipulate it atomically, which are:

- "Down" or "P": wait for the value to become positive, then decrement it.
- "Up" or "V": increment the value (and wake up one waiting thread, if any).

A semaphore initialized to 0 may be used to wait for an event that will happen exactly once. For example, suppose thread `A` starts another thread `B` and wants to wait for `B` to signal that some activity is complete. `A` can create a semaphore initialized to 0, pass it to `B` as it starts it, and then "down" the semaphore. When B finishes its activity, it "ups" the semaphore. This works regardless of whether A "downs" the semaphore or B "ups" it first.

A semaphore initialized to 1 is typically used for controlling access to a resource. Before a block of code starts using the resource, it "downs" the semaphore, then after it is done with the resource it "ups" the resource. In such a case a lock, described below, may be more appropriate.

Semaphores can also be initialized to 0 or values larger than 1.

Pintos' semaphore type and operations are declared in `threads/synch.h`.

- **Type:** `struct semaphore`
  Represents a semaphore.
- **Function:** `void sema_init (struct semaphore *sema, unsigned value)`
  Initializes `sema` as a new semaphore with the given initial `value`.
- **Function:** `void sema_down (struct semaphore *sema)`
  Executes the "down" or "P" operation on `sema`, waiting for its value to become positive and then decrementing it by one.
- **Function:** `bool sema_try_down (struct semaphore *sema)`
  Tries to execute the "down" or "P" operation on `sema`, without waiting. Returns true if `sema` was successfully decremented, or false if it was already zero and thus could not be decremented without waiting. Calling this function in a tight loop wastes CPU time, so use `sema_down` or find a different approach instead.
- **Function:** `void sema_up (struct semaphore *sema)`
  Executes the "up" or "V" operation on `sema`, incrementing its value. If any threads are waiting on `sema`, wakes one of them up.

  Unlike most synchronization primitives, `sema_up` may be called inside an external interrupt handler.

Semaphores are internally built out of disabling interrupt and thread blocking and unblocking (`thread_block` and `thread_unblock`). Each semaphore maintains a list of waiting threads, using the linked list implementation in `lib/kernel/list.c`.

3.3.3 Locks

A **lock** is like a semaphore with an initial value of 1. A lock's equivalent of "up" is called "release", and the "down" operation is called "acquire".

Compared to a semaphore, a lock has one added restriction: only the thread that acquires a lock, called the lock's "owner", is allowed to release it. If this restriction is a problem, it's a good sign that a semaphore should be used, instead of a lock.

Locks in Pintos are not "recursive," that is, it is an error for the thread currently holding a lock to try to acquire that lock.

Lock types and functions are declared in `threads/synch.h`.

- **Type:** `struct lock`
  Represents a lock.
- **Function:** `void lock_init (struct lock *lock)`
  Initializes `lock` as a new lock. The lock is not initially owned by any thread.
- **Function:** `void lock_acquire (struct lock *lock)`
  Acquires `lock` for the current thread, first waiting for any current owner to release it if necessary.

- **Function:** `bool lock_try_acquire (struct lock *lock)`
  Tries to acquire `lock` for use by the current thread, without waiting. Returns true if successful, false if the lock is already owned. Calling this function in a tight loop is a bad idea because it wastes CPU time, so use `lock_acquire` instead.

- **Function:** `void lock_release (struct lock *lock)`
  Releases `lock`, which the current thread must own.

- **Function:** `bool lock_held_by_current_thread (const struct lock *lock)`
  Returns true if the running thread owns `lock`, false otherwise. There is no function to test whether an arbitrary thread owns a lock, because the answer could change before the caller could act on it.

3.3.4 Monitors

A **monitor** is a higher-level form of synchronization than a semaphore or a lock. A monitor consists of data being synchronized, plus a lock, called the **monitor lock**, and one or more **condition variables**. Before it accesses the protected data, a thread first acquires the monitor lock. It is then said to be "in the monitor". While in the monitor, the thread has control over all the protected data, which it may freely examine or modify. When access to the protected data is complete, it releases the monitor lock.

Condition variables allow code in the monitor to wait for a condition to become true. Each condition variable is associated with an abstract condition, e.g. "some data has arrived for processing" or "over 10 seconds has passed since the user's last keystroke". When code in the monitor needs to wait for a condition to become true, it "waits" on the associated condition variable, which releases the lock and waits for the condition to be signaled. If, on the other hand, it has caused one of these conditions to become true, it "signals" the condition to wake up one waiter, or "broadcasts" the condition to wake all of them.

The theoretical framework for monitors was laid out by C. A. R. Hoare. Their practical usage was later elaborated in a paper on the Mesa operating system.

Condition variable types and functions are declared in `threads/synch.h`.

- **Type:** `struct condition`
  Represents a condition variable.

- **Function:** `void cond_init (struct condition *cond)`
  Initializes `cond` as a new condition variable.

- **Function:** `void cond_wait (struct condition *cond, struct lock *lock)`
  Atomically releases `lock` (the monitor lock) and waits for `cond` to be signaled by some other piece of code. After `cond` is signaled, reacquires `lock` before returning. `lock` must be held before calling this function.

  Sending a signal and waking up from a wait are not an atomic operation. Thus, typically `cond_wait`'s caller must recheck the condition after the wait completes and, if necessary, wait again.

- **Function:** `void cond_signal (struct condition *cond, struct lock *lock)`
  If any threads are waiting on `cond` (protected by monitor lock `lock`), then this function wakes up one of them. If no threads are waiting, returns without performing any action. `lock` must be held before calling this function.

- **Function:** `void cond_broadcast (struct condition *cond, struct lock *lock)`
  Wakes up all threads, if any, waiting on `cond` (protected by monitor lock `lock`). `lock` must be held before calling this function.

3.3.5 Optimization Barriers

An **optimization barrier** is a special statement that prevents the compiler from making assumptions about the state of memory across the barrier. The compiler will not reorder reads or writes of variables across the barrier or assume that a variable's value is unmodified across the barrier, except for local variables whose address is never taken. In Pintos, `threads/synch.h` defines the `barrier()` macro as an optimization barrier.

One reason to use an optimization barrier is when data can change asynchronously, without the compiler's knowledge, e.g. by another thread or an interrupt handler. The `too_many_loops` function in `devices/timer.c` is an example. This function starts out by busy-waiting in a loop until a timer tick occurs:

```
/* Wait for a timer tick. */
int64_t start = ticks;
while (ticks == start)
  barrier ();
```

Without an optimization barrier in the loop, the compiler could conclude that the loop would never terminate, because `start` and `ticks` start out equal and the loop itself never changes them. It could then "optimize" the function into an infinite loop, which would definitely be undesirable.

Optimization barriers can be used to avoid other compiler optimizations. The `busy_wait` function, also in `devices/timer.c`, is an example. It contains this loop:

```
while (loops-- > 0)
  barrier ();
```

The goal of this loop is to busy-wait by counting `loops` down from its original value to 0. Without the barrier, the compiler could delete the loop entirely, because it produces no useful output and has no side effects. The barrier forces the compiler to pretend that the loop body has an important effect.

Finally, optimization barriers can be used to force the ordering of memory reads or writes. For example, suppose we add a "feature" that, whenever a timer interrupt occurs, the character in global variable `timer_put_char` is printed on the console, but only if global Boolean variable `timer_do_put` is true. The best way to set up `x` to be printed is then to use an optimization barrier, like this:

```
timer_put_char = 'x';
barrier ();
timer_do_put = true;
```

Without the barrier, the code is buggy because the compiler is free to reorder operations when it doesn't see a reason to keep them in the same order. In this case, the compiler doesn't know that the order of assignments is important, so its optimizer is permitted to exchange their order. There's no telling whether it will actually do this, and it is possible that passing the compiler different optimization flags or using a different version of the compiler will produce different behavior.

Another solution is to disable interrupts around the assignments. This does not prevent reordering, but it prevents the interrupt handler from intervening between the assignments. It also has the extra runtime cost of disabling and re-enabling interrupts:

```
enum intr_level old_level = intr_disable ();
timer_put_char = 'x';
timer_do_put = true;
intr_set_level (old_level);
```

A second solution is to mark the declarations of `timer_put_char` and `timer_do_put` as `volatile`. This keyword tells the compiler that the variables are externally observable and restricts its latitude for optimization. However, the semantics of `volatile` are not well-defined, so it is not a good general solution. The base Pintos code does not use `volatile` at all.

The following is *not* a solution, because locks neither prevent interrupts nor prevent the compiler from reordering the code within the region where the lock is held:

```
lock_acquire (&timer_lock);     /* INCORRECT CODE */
timer_put_char = 'x';
timer_do_put = true;
lock_release (&timer_lock);
```

The compiler treats invocation of any function defined externally, that is, in another source file, as a limited form of optimization barrier. Specifically, the compiler assumes that any externally defined function may access any statically or dynamically allocated data and any local variable whose address is taken. This often means that explicit barriers can be omitted. It is one reason that Pintos contains few explicit barriers.

A function defined in the same source file, or in a header included by the source file, cannot be relied upon as an optimization barrier. This applies even to invocation of a function before its definition, because the compiler may read and parse the entire source file before performing optimization.

3.4 Memory Allocation

Pintos contains two memory allocators, one that allocates memory in units of a page, and one that can allocate blocks of any size.

3.4.1 Page Allocator

The page allocator declared in `threads/palloc.h` allocates memory in units of a page. It is most often used to allocate memory one page at a time, but it can also allocate multiple contiguous pages at once.

The page allocator divides the memory it allocates into two pools, called the kernel and user pools. By default, each pool gets half of system memory above 1 MiB, but the division can be changed with the `-ul` kernel command line option. An allocation request draws from one pool or the other. If one pool becomes empty, the other may still have free pages. The user pool should be used for allocating memory for user processes and the kernel pool for all other allocations. This distinction is not very relevant in this project, since all threads you will be dealing with are kernel threads (unlike in Project 1). For Project 2, all allocations should be made from the kernel pool.

Each pool's usage is tracked with a bitmap, one bit per page in the pool. A request to allocate `n` pages scans the bitmap for `n` consecutive bits set to false, indicating that those pages are free, and then sets those bits to true to mark them as used. This is a "first fit" allocation strategy.

The page allocator is subject to fragmentation. That is, it may not be possible to allocate `n` contiguous pages even though `n` or more pages are free, because the free pages are separated by used pages. In fact, in pathological cases it may be impossible to allocate 2 contiguous pages even though half of the pool's pages are free. Single-page requests can't fail due to fragmentation, so requests for multiple contiguous pages should be limited as much as possible.

Pages may not be allocated from interrupt context, but they may be freed.

When a page is freed, all of its bytes are cleared to `0xcc`, as a debugging aid.

Page allocator types and functions are described below.

- **Function:** `void * palloc_get_page (enum palloc_flags flags)`
  Function: `void * palloc_get_multiple (enum palloc_flags flags, size_t page_cnt)`

Obtains and returns one page, or `page_cnt` contiguous pages, respectively. Returns a null pointer if the pages cannot be allocated.

The `flags` argument may be any combination of the following flags:

- **Page Allocator Flag:** `PAL_ASSERT`
  If the pages cannot be allocated, panic the kernel. This is only appropriate during kernel initialization. User processes should never be permitted to panic the kernel.
- **Page Allocator Flag:** `PAL_ZERO`
  Zero all the bytes in the allocated pages before returning them. If not set, the contents of newly allocated pages are unpredictable.
- **Page Allocator Flag** `PAL_USER`
  Obtain the pages from the user pool. If not set, pages are allocated from the kernel pool.

- **Function:** `void palloc_free_page (void *page)`
  Function: `void palloc_free_multiple (void *pages, size_t page_cnt)`
  Frees one page, or `page_cnt` contiguous pages, respectively, starting at `pages`. All of the pages must have been obtained using `palloc_get_page` or `palloc_get_multiple`.

3.4.2 Block Allocator

The block allocator, declared in `threads/malloc.h`, can allocate blocks of any size. It is layered on top of the page allocator described in the previous section. Blocks returned by the block allocator are obtained from the kernel pool.

The block allocator uses two different strategies for allocating memory. The first strategy applies to blocks that are 1 KiB or smaller (one-fourth of the page size). These allocations are rounded up to the nearest power of 2, or 16 bytes, whichever is larger. Then they are grouped into a page used only for allocations of that size.

The second strategy applies to blocks larger than 1 KiB. These allocations (plus a small amount of overhead) are rounded up to the nearest page in size, and then the block allocator requests that number of contiguous pages from the page allocator.

In either case, the difference between the allocation requested size and the actual block size is wasted. A real operating system would carefully tune its allocator to minimize this waste, but this is unimportant in an instructional system like Pintos.

As long as a page can be obtained from the page allocator, small allocations always succeed. Most small allocations do not require a new page from the page allocator at all, because they are satisfied using part of a page already allocated. However, large allocations always require calling into the page allocator, and any allocation that needs more than one contiguous page can fail due to fragmentation, as already discussed in the previous section. Thus, you should minimize the number of large allocations in your code, especially those over approximately 4 KiB each.

When a block is freed, all of its bytes are cleared to `0xcc`, as a debugging aid.

The block allocator may not be called from interrupt context.

The block allocator functions are described below. Their interfaces are the same as the standard C library functions of the same names.

- **Function:** `void * malloc (size_t size)`
  Obtains and returns a new block, from the kernel pool, at least `size` bytes long. Returns a null pointer if `size` is zero or if memory is not available.

- **Function:** `void * calloc (size_t a, size_t b)`
  Obtains a returns a new block, from the kernel pool, at least `a * b` bytes long. The block's contents will be cleared to zeros. Returns a null pointer if `a` or `b` is zero or if insufficient memory is available.

- **Function:** `void * realloc (void *block, size_t new_size)`
  Attempts to resize `block` to `new_size` bytes, possibly moving it in the process. If successful, returns the new block, in which case the old block must no longer be accessed. On failure, returns a null pointer, and the old block remains valid.

  A call with `block` null is equivalent to `malloc`. A call with `new_size` zero is equivalent to `free`.

- **Function:** `void free (void *block)`
  Frees `block`, which must have been previously returned by `malloc`, `calloc`, or `realloc` (and not yet freed).

3.5 Linked Lists

Pintos contains a linked list data structure in `lib/kernel/list.h` that is used for many different purposes. This linked list implementation is different from most other linked list implementations you may have encountered, because **it does not use any dynamic memory allocation**.

```
/* List element. */
struct list_elem
  {
    struct list_elem *prev;     /* Previous list element. */
    struct list_elem *next;     /* Next list element. */
  };

/* List. */
struct list
  {
    struct list_elem head;      /* List head. */
    struct list_elem tail;      /* List tail. */
  };
```

In a Pintos linked list, each list element contains a "`struct list_elem`", which contains the pointers to the next and previous element. Because the list elements themselves have enough space to hold the prev and next pointers, we don't need to allocate any extra space to support our linked list. Here is an example of a linked list element which can hold an integer:

```
/* Integer linked list */
struct int_list_elem
  {
    int value;
    struct list_elem elem;
  };
```

Next, you must create a "`struct list`" to represent the whole list. Initialize it with `list_init()`.

```
/* Declare and initialize a list */
struct list my_list;
list_init (&my_list);
```

Now, you can declare a list element and add it to the end of the list. Notice that the second argument of `list_push_back()` is the address of a "`struct list_elem`", not the "`struct int_list_elem`" itself.

```
/* Declare a list element. */
struct int_list_elem three = {3, {NULL, NULL}};

/* Add it to the list */
list_push_back (&my_list, &three.elem);
```

We can use the `list_entry()` macro to convert a generic "`struct list_elem`" into our custom "`struct int_list_elem`" type. Then, we can grab the "value" attribute and print it out:

```
/* Fetch elements from the list */
struct list_elem *first_list_element = list_begin (&my_list);
struct int_list_elem *first_integer = list_entry (first_list_element,
                                                  struct int_list_elem,
                                                  elem);
printf("The first element is: %d\n", first_integer->value);
```

By storing the prev and next pointers inside the structs themselves, we can avoid creating new "linked list element" containers. However, this also means that a `list_elem` can only be part of one list a time. Additionally, our list should be homogeneous (it should only contain one type of element).

The `list_entry()` macro works by computing the offset of the `elem` field inside of "`struct int_list_elem`". In our example, this offset is 4 bytes. To convert a pointer to a generic "`struct list_elem`" to a pointer to our custom "`struct int_list_elem`", the `list_entry()` just needs to subtract 4 bytes! (It also casts the pointer, in order to satisfy the C type system.)

Linked lists have 2 sentinel elements: the `head` and `tail` elements of the "`struct list`". These sentinel elements can be distinguished by their `NULL` pointer values. Make sure to distinguish between functions that return the first actual element of a list and functions that return the sentinel `head` element of the list.

There are also functions that sort a link list (using quicksort) and functions that insert an element into a sorted list. These functions require you to provide a list element comparison function (see `lib/kernel/list.h` for more details).

3.6 Efficient Alarm Clock

Here are some more details about the Efficient Alarm Clock task.

1. If `timer_sleep()` is called with a zero or negative argument, then you should just return immediately.
2. When you run Pintos, the clock does **not** run in realtime by default. So, if a thread goes to sleep for 5 "seconds" (e.g. ticks = 5 × TIMER_FREQ), it will actually be much shorter than 5 seconds in terms of wall clock time. You can use the `--realtime` flag for Pintos to override this.
3. 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.
4. The code that runs in interrupt handlers (i.e. `timer_interrupt()`) should be as fast as possible. It's usually wise to do some pre-computation outside of the interrupt handler, in order to make the interrupt handler as fast as possible. Additionally, you may not acquire locks while executing `timer_interrupt()`.
5. Pay close attention to the Pintos linked-list implementation. Each linked list requires a dedicated `list_elem` member inside its elements. Every element of a linked list should be the same type. If you create new linked lists, make sure that they are initialized. Finally, make sure that there are no race conditions for any of your linked lists (the list manipulation functions are **NOT** thread-safe).

3.7 Priority Scheduler

Here are some more details about the Priority Scheduler task.

1. A thread's initial priority is an argument of `thread_create()`. You should use `PRI_DEFAULT` (31), unless there is a reason to use a different value.

2. Your implementation must handle nested donation: Consider a high-priority thread H, a medium-priority thread M, and a low-priority thread L. If H must wait on M and M must wait on L, then we should donate H's priority to L.

3. A thread can only donate to 1 thread at a time, because once it calls `lock_acquire()`, the donor thread is blocked.

4. If there are multiple waiters on a lock when you call `lock_release()`, then all of those priority donations must apply to the thread that receives the lock next.

5. You do not need to handle priority values outside of the allowed range, `PRI_MIN` (0) to `PRI_MAX` (63).

6. You only need to implement priority donation for locks. Do not implement them for other synchronization variables (it doesn't make any sense to do it for semaphores or monitors anyway). However, you need to implement priority scheduling for locks, semaphores, and condition variables. Priority scheduling is when you unblock the highest priority thread when a resource is released or a monitor is signaled.

7. Don't forget to implement `thread_get_priority()`, which is the function that returns the current thread's priority. This function should take donations into account. You should return the **effective priority** of the thread.

8. A thread cannot change another thread's priority, except via donations. The `thread_set_priority()` function only acts on the current thread.

9. If a thread no longer has the highest effective priority (e.g. because it released a lock or it called `thread_set_priority()` with a lower value), it must immediately yield the CPU. If a lock is released, but the current thread still has the highest effective priority, it should not yield the CPU.

3.8 Advanced Scheduler (Optional)

Here are some more details about the **optional** Advanced Scheduler task.

3.8.1 Introduction

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 this task, you must implement the scheduler according to the specification in this section. However, the exact method of implementation is up to you. As long as the behavior of your scheduler matches the specification here, it is acceptable.

Multiple parts of this 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.

When the advanced scheduler is enabled, you should **NOT** do priority donation.

3.8.2 Fixed-point Real Numbers

Many of the calculations in the following section assume that you're using real numbers, not integers. However, Pintos does not support floating point number operations. We have provided the `fixed-point.h` library inside `pintos/src/threads/fixed-point.h`, which will allow you to use fixed point numbers to represent real numbers. You should use `fixed_point_t` and the functions defined in `fixed-point.h` to represent any value that needs to be a real number. If you use integers, your values will not be correct.

3.8.3 Niceness

Each thread 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 the test framework. We have provided skeleton definitions for them in "`threads/thread.c`".

- **Function:** `int thread_get_nice (void)`
  Returns the current thread's `nice` value.

- **Function:** `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. If the running thread no longer has the highest priority, it should yield the CPU.

3.8.4 Calculating Priority

Our scheduler has 64 priorities 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. The scheduler should always choose the highest priority thread to run next. If there are multiple threads with the highest priority, then the scheduler should cycle through each of these threads in "round robin" fashion. 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

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

In this formula, `recent_cpu` is an estimate of the CPU time the thread has used recently (see the next section on `recent_cpu`) 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 is designed so that threads that have recently been scheduled on the CPU will have a lower priority the next time the scheduler picks a thread to run. 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 very high `nice` value, should ensure that it receives CPU time soon.

3.8.5 Calculating Recent CPU

We wish recent_cpu to measure how much CPU time each process has received "recently." 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:

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

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

$$x(1) = f(1)$$
$$x(2) = a \times f(1) + f(2)$$
$$x(3) = a^2 \times f(1) + a \times f(2) + f(3)$$
$$x(4) = a^3 \times f(1) + a^2 \times f(2) + a \times f(3) + f(4)$$

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 $t+2 \times k$, 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:

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

In this formula, load_avg is a moving average of the number of threads ready to run (see the next section). 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 $ln(0.1)/ln(\frac{3}{4}) = 0.8$ 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".

- **Function:** `int thread_get_recent_cpu(void)`
  Returns 100 times the current thread's recent_cpu value, rounded to the nearest integer.

3.8.6 Calculating Load Average

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:

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

In this formula, `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`".

- **Function:** `int thread_get_load_avg(void)`
  Returns 100 times the current system load average, rounded to the nearest integer.

3.8.7 Summary

The following formulas summarize the calculations required to implement the scheduler.

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:

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

`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:

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

`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:

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

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

3.8.8 Additional Details

1. When the advanced scheduler is enabled, you should **NOT** do priority donation.

2. When the advanced 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.

3. Because many of these formulas involve fractions, you should use **fixed-point real arithmetic** for your calculations. Use the `fixed_point_t` type and the library functions inside `pintos/src/threads/fixed-point.h` to do your advanced scheduler calculations.

3.9 Debugging Tips

We discussed a variety of debugging tools in the specification for Project 1. To demonstrate how to use them in the context of this project, we've included a sample GDB session below.

Sample GDB Session This section narrates a sample GDB session, provided by Godmar Back. This example illustrates how one might debug a project 1 solution in which occasionally a thread that calls `timer_sleep` is not woken up. With this bug, tests such as `mlfqs_load_1` get stuck.

This session was captured with a slightly older version of Bochs and the gdb macros for Pintos, so it looks slightly different than it would now.

First, I start Pintos:

```
 $ pintos -v --gdb -- -q -mlfqs run mlfqs-load-1

writing command line to /tmp/gdalqtb5uf.dsk...
Bochs -q
========================================================================
Bochs x86 emulator 2.2.5
build from cvs snapshot on december 30, 2005
========================================================================
00000000000i[     ] reading configuration from Bochsrc.txt
00000000000i[     ] enabled gdbstub
00000000000i[     ] installing nogui module as the Bochs gui
00000000000i[     ] using log file Bochsout.txt
waiting for gdb connection on localhost:1234
```

Then, I open a second window on the same machine and start gdb:

```
$ pintos-gdb kernel.o

gnu gdb red hat linux (6.3.0.0-1.84rh)
copyright 2004 free software foundation, inc.
gdb is free software, covered by the gnu general public license, and you are
welcome to change it and/or distribute copies of it under certain conditions.
type "show copying" to see the conditions.
there is absolutely no warranty for gdb.  type "show warranty" for details.
this gdb was configured as "i386-redhat-linux-gnu"...
using host libthread_db library "/lib/libthread_db.so.1".
```

Then, I tell gdb to attach to the waiting Pintos emulator:

```
(gdb) debugpintos
remote debugging using localhost:1234
0x0000fff0 in ?? ()
reply contains invalid hex digit 78
```

Now I tell Pintos to run by executing `c` (short for `continue`):

Now Pintos will continue and output:

```
pintos booting with 4,096 kb ram...
kernel command line: -q -mlfqs run mlfqs-load-1
374 pages available in kernel pool.
```

```
373 pages available in user pool.
calibrating timer...  102,400 loops/s.
boot complete.
executing 'mlfqs-load-1':
(mlfqs-load-1) begin
(mlfqs-load-1) spinning for up to 45 seconds, please wait...
(mlfqs-load-1) load average rose to 0.5 after 42 seconds
(mlfqs-load-1) sleeping for another 10 seconds, please wait...
```

Until it gets stuck because of the bug I had introduced. I hit `ctrl+c` in the debugger window:

```
program received signal 0, signal 0.
0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
649    while (i <= pri_max && list_empty (&ready_list[i]))
(gdb)
```

The thread that was running when I interrupted Pintos was the idle thread. If i run `backtrace`, it shows this backtrace:

```
(gdb) bt
#0  0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
#1  0xc0101778 in schedule () at ../../threads/thread.c:714
#2  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#3  0xc0101419 in idle (aux=0x0) at ../../threads/thread.c:551
#4  0xc010145a in kernel_thread (function=0xc01013ff , aux=0x0)
at ../../threads/thread.c:575
#5  0x00000000 in ?? ()
```

Not terribly useful. What I really like to know is what's up with the other thread (or threads). Since I keep all threads in a linked list called `all_list`, linked together by a `struct list_elem` member named `allelem`, I can use the `btthreadlist` macro from the macro library I wrote. `btthreadlist` iterates through the list of threads and prints the backtrace for each thread:

```
(gdb) btthreadlist &all_list allelem
pintos-debug: dumping backtrace of thread 'main' @@0xc002f000
#0  0xc0101820 in schedule () at ../../threads/thread.c:722
#1  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#2  0xc0104755 in timer_sleep (ticks=1000) at ../../devices/timer.c:141
#3  0xc010bf7c in test_mlfqs_load_1 () at ../../tests/threads/mlfqs-load-1.c:49
#4  0xc010aabb in run_test (name=0xc0007d8c "mlfqs-load-1")
at ../../tests/threads/tests.c:50
#5  0xc0100647 in run_task (argv=0xc0110d28) at ../../threads/init.c:281
#6  0xc0100721 in run_actions (argv=0xc0110d28) at ../../threads/init.c:331
#7  0xc01000c7 in main () at ../../threads/init.c:140

pintos-debug: dumping backtrace of thread 'idle' @@0xc0116000
#0  0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
#1  0xc0101778 in schedule () at ../../threads/thread.c:714
#2  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#3  0xc0101419 in idle (aux=0x0) at ../../threads/thread.c:551
```

```
#4  0xc010145a in kernel_thread (function=0xc01013ff , aux=0x0)
at ../../threads/thread.c:575
#5  0x00000000 in ?? ()
```

In this case, there are only two threads, the idle thread and the main thread. The kernel stack pages (to which the `struct thread` points) are at `0xc0116000` and verb—0xc002f000—, respectively. The main thread is stuck in `timer_sleep`, called from `test_mlfqs_load_1`.

Knowing where threads are stuck can be tremendously useful, for instance when diagnosing deadlocks or unexplained hangs.