CS 162 Project 2: `Threads`

Design Document Due:	Friday, October 15, 2021
[Ungraded] Checkpoint Due:	Sunday, October 24, 2021
Code Due:	Friday, November 05, 2021
Final Report Due:	Friday, November 05, 2021

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. Importantly, each thread that was also a userspace process in Project 1 was the *only* thread in that process; multithreaded user programs were not supported. Task 3: User Threads of this project, we will be overcoming this limitation by adding support for multithreaded user programs. For Task 1: Efficient Alarm Clock and Task 2: Schedulers of this project, we will simplify things by implementing features only for kernel threads—threads that only execute in the kernel mode and have no userspace component.

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: Schedulers

1.2.1 Strict 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.

1.2.2 Fair Priority Scheduler

You may remember from lecture that few modern operating systems use truly strict priority schedulers; this is chiefly due to their propensity for inducing starvation in low-priority threads. Rather than simply getting proportionally less time on the CPU, threads with low enough priority will often fail to ever get scheduled at all. As such, *in addition* to the strict priority scheduler algorithm described in Strict Priority Scheduler, you must design and implement a second, priority-based scheduler, built around an algorithm of your choice selected to ensure that all threads are able to make consistent, if slow, forward progress. For more details on exactly what is expected of your implementation for this component, refer to the Fair Scheduler section.

This section is meant to be design-heavy, and as such we expect your design document to be especially thorough with regards to this component. We will expect you to have considered a minimum of three possible approaches to this problem, so take care to describe:

I. **Which** options you considered (minimum of three);

II. **What** relative advantages and disadvantages each has;

III. **Why** you chose the algorithm you did from among them.

Your writeup for this task will constitute a sizeable portion of your design document grade. Note that your final algorithm does not have to be particularly complex, and you will not be graded on the intricacy of your design: to the contrary, many of the best schedulers in use today are deceptively simple at their core.

We look forward to seeing what you come up with!

1.3 Task 3: User Threads

Pintos is a multithreaded kernel, i.e. there can be more than one thread running in the kernel. While working on Project 1, you have no doubt worked with the threading interface in the kernel. In `threads/thread.c`, the `thread_create` function allows us to create a new kernel thread that runs a specific kernel task, and the `thread_exit` function allows a thread to kill itself. You should read and understand the kernel threading model.

On the other hand, as it were in Project 1, each user process only had one thread of control. In other words, it is impossible for a user program to create a new thread to run another user function – there was no analog of `thread_create` and `thread_exit` for user programs. In a real system, user programs can indeed create their own threads. We saw this via the `pthread` library, which we learned about in discussion[1].

In this project, you will need to implement a simplified version of the `pthread` library that allows user programs to create their own threads using the functions `pthread_create` and `pthread_exit` in `lib/user/pthread.h`. Threads can also wait on other threads with the `pthread_join` function, which is similar to the `wait` system call for processes. Threads should be able to learn their thread IDs (TIDs) through a new `get_tid` system call. You must also account for how the syscalls in Project 1 are affected by making user programs multithreaded.

- In Project 1, whenever a user program (which consisted of just a single thread) trapped into the OS, it ran in its own dedicated kernel thread. In other words, user threads had a 1-1 mapping with kernel threads. For Task 3: User Threads, you will need to maintain this 1-1 mapping; that is, a user process with `n` user threads should be paired 1-1 with `n` kernel threads, and each user thread should run in its dedicated kernel thread when it traps into the OS.

[1] https://cs162.org/static/sections/section1-sol.pdf

- You should **not** implement green threads[2], which have a many-to-one mapping between user threads and kernel threads. Green threads are not ideal, because as soon as one user thread blocks, e.g. on IO, all of the user threads are also blocked.

In addition, you must also implement user-level synchronization. After all, threads are not all that useful if we can't synchronize them properly with locks and semaphores. You will be required to implement `lock_init`, `lock_acquire`, `lock_release`, `sema_init`, `sema_down`, and `sema_up` for user programs.

[2]https://en.wikipedia.org/wiki/Green_threads

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

3. Suppose there are two kernel threads in the system, Thread A running `functionA` and Thread B running `functionB`. Give a scheduler ordering in which the following code can lead to deadlock.

```
struct lock lockA; // Global lock
struct lock lockB; // Global lock

void functionA() {
  lock_acquire(&lockA);
  lock_acquire(&lockB);
  lock_release(&lockB);
  lock_release(&lockA);
}

void functionB() {
  lock_acquire(&lockB);
  lock_acquire(&lockA);
  lock_release(&lockA);
  lock_release(&lockB);
}
```

4. Consider the following scenario: there are two kernel threads in the system, Thread A and Thread B. Thread A is running in the kernel, which means Thread B must be on the ready queue, waiting patiently in `threads/switch.S`. Currently in Pintos, threads cannot forcibly kill each other. But suppose that Thread A decides to kill Thread B by taking it off the ready queue and freeing its thread stack. This will prevent Thread B from running, but what issues could arise later from this action?

5. Consider a fully-functional correct implementation of this project, except for a single bug, which exists in the kernel's `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

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.

2.3 Checkpoint [Ungraded]

You should have implemented Task 1: Efficient Alarm Clock by the checkpoint deadline. Keep in mind that both Strict Priority Scheduler and Task 3: User Threads are significantly more time-consuming to implement, so you may wish to begin Task 2 and Task 3 by this date even though it is not part of this checkpoint.

2.4 Final Code

You must have completed all coding tasks (Task 1: Efficient Alarm Clock, Task 2: Schedulers, and Task 3: User Threads) in their entirety.

2.5 Final Report 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 scheduler implementation follow best practices for kernel development, particularly with regards to efficiency?
- 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 both *kernel threads*, that is, threads that operate in the kernel without any userspace component, as well as *user threads*, which are threads that operate in userspace and can *trap* to the kernel on interrupts, system calls, or exceptions.

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.

We recommend that you first use Git to tag your final Project 1 code, for your own benefit. You may end up 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. By the end of the project, your code should pass all of the tests.

Many of the tests are the same as those from Project 1, because we want to make sure your changes to Project 2 do not break your changes to Project 1. However, not all of the tests will be equally weighted.

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 Task 2: Schedulers 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.

`userprog/`
Implementation of user programs that you modified for Project 1. You will need to update the syscall interface to add support for multithreading in Task 3: User Threads.

`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.)

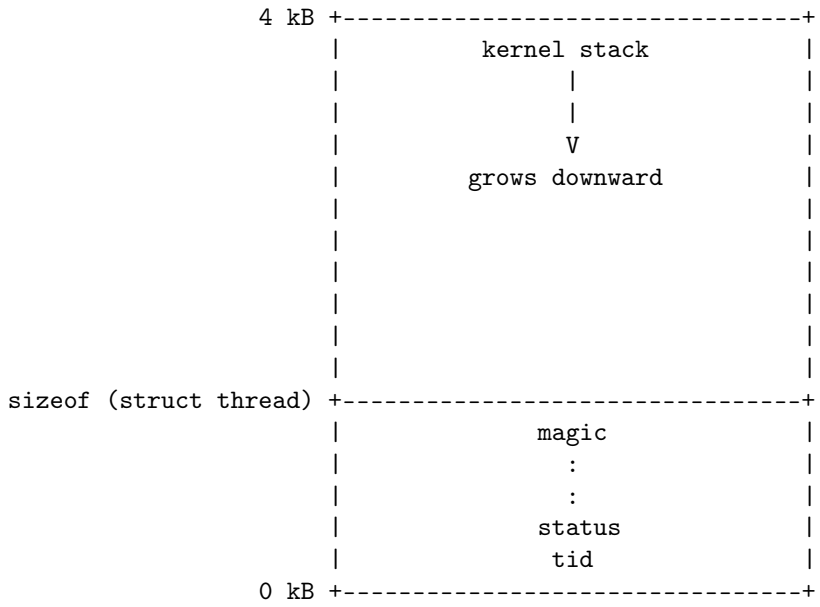
The mechanics of a context switch are in `threads/switch.S`, which is x86 assembly code. It saves the state of the currently running thread and restores the state of the next thread onto the CPU.

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:



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:** `struct process* pcb`
  (Used in Task 3: User Threads.) The process control block (PCB) for this process, if this thread belongs to 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.

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 Processes

3.3.1 Processes Overview

Recall that in a multithreaded user process, each user thread has an associated kernel thread and a thread control block (TCB), and the process as a whole shares a process control block (PCB). The TCB contains information relevant to a particular thread, like its name, priority, or its stack pointer, whereas the PCB contains information relevant to every thread in the process, like the file descriptor table.

In Pintos, the TCB of a thread is stored within the same page as the thread's stack, at the bottom of the page. In Project 1 when user processes were single threaded, it was possible to embed the PCB within the TCB, since no other threads would need to access it. So in Project 1, the PCB was contained inside `struct thread` within an `#ifdef USERPROG` statement.

However, in order to implement user-level mulithreading, each of the kernel threads that back the user threads must share a PCB, we can no longer embed the PCB within the TCB. (We could try to embed the TCB within the 'main' thread's TCB, but this is messy and wastes space on the thread stack). Instead, each process has to `malloc` a separate struct, called `struct process`, which contains the PCB for the process. Each thread has a pointer to the PCB. The PCB is allocated when the process is created, and free'd when the process exits.

To reduce implementation overhead for Task 3: User Threads, within the starter code we have already refactored the PCB to be a separate struct instead of embedded within the TCB. You can see the struct definition in `userprog/process.h` and see how it is created and destroyed in `userprog/process.c`. However, our refactor is barebones – you may find it necessary to add members to both `struct process` and `struct thread` to get a fully functional implementation.

3.3.2 The Process Struct

The full definition of `struct process` is copied below for your convenience.

```
/* The process control block for a given process. Since
   there can be multiple threads per process, we need a separate
   PCB from the TCB. All TCBs in a process will have a pointer
   to the PCB, and the PCB will have a pointer to the main thread
   of the process, which is 'special'. */
struct process {
  /* Owned by process.c. */
  struct wait_status* wait_status; /* This process's completion status. */
  struct list children;            /* Completion status of children. */
  uint32_t* pagedir;               /* Page directory. */
  char process_name[16];           /* Name of the main thread */
  struct file* bin_file;           /* Executable. */
  struct thread* main_thread;      /* Pointer to main thread */

  /* Owned by syscall.c. */
  struct list fds; /* List of file descriptors. */
  int next_handle; /* Next handle value. */
};
```

- **Member of `struct process` `struct wait_status* wait_status`**

  A pointer to a struct that contains information about this process's completion status. The dynamially allocated `wait_status` struct contains a semaphore allowing the parent of a process to `wait` on its child, and it contains the exit code of the child process for the parent. The full definition of `struct wait_status` can be found in `userprog/process.h`.

- **Member of `struct process` `struct list children`**

  A list of all the completion statuses of the children of this process.

- **Member of `struct process` `uint32_t* pagedir`**

  A pointer to the page table for this process

- **Member of** `struct process` `char process_name[16]`

  The name of a process. This is simply the name of the process's main thread.

- **Member of** `struct process` `struct file* bin_file`

  A pointer to the executable file from which this process was loaded.

- **Member of** `struct process` `struct thread* main_thread`

  A pointer to the "main thread" of a process. Every user program starts as single-threaded, and this single thread is referred to as the "main thread" of the process. Once user-level multithreading is implemented, the main thread can create more threads using the functions in `lib/user/pthread.h`.

- **Member of** `struct process` `struct list fds`

  A list of all the file-descriptor to `struct file*` mappings for this user process. These are encoded in a `struct file_descriptor` whose definition is available in `userprog/process.h`

- **Member of** `struct process` `int next_handle`

  The next file descriptor number to allocate to a user process. The only requirement is that these are unique for unique resources, so this number is continuously increasing.

3.3.3 Processes Details

The First User Program All user programs are spawned from other user programs using the `exec` system call. So how is the first user program created? When the operating system starts up, it first runs `threads/init.c`. If the command-line arguments to Pintos indicate that the user wants to start running user programs, the `run_task` function is called, which has the main thread of Pintos (the one running the `threads/init.c` code) call `process_wait(process_execute(task))`. So, the first user program is created by tha main thread of the OS.

Quick sidebar: it's important to disambiguate between the "main" thread of the OS and the main thread of a user program. The main thread of Pintos is the thread that runs `threads/init.c` – it is the thread that sets up the OS and starts running the first task, whether that is a user program or a kernel task. The type of task is dependent on the command-line arguments to Pintos. On the other hand, the main thread of a user program is the single thread that is running when that user program is first created. It should be clear from context which "main" thread we are referring to.

Because the first user program is created by the OS's main thread, the OS's main thread must have a PCB, even though it is not a process and will never run user-level code. This is because our implementation of `process_wait` and `process_execute` require the parent "process" to have access to a list of all child process' completion statuses. So, in the `userprog_init` function, we give the OS's main thread a (minimal) PCB so that it can successfully execute those functions. Right now, it's only possible that the main thread tries to access the `children` list in those functions. However, if you modify those functions so that the main thread must access another member of `struct process`, you must initialize that member in `userprog_init` in `userprog/process.c`.

PIDs and TIDs In the starter code, we define the process ID (PID) of a process to be the thread ID (TID) of its main thread. The following two functions in `userprog/process.c` are related:

- `bool is_main_thread(struct thread* t, struct process* p)`

  Returns true if t is the main thread of p

- `pid_t get_pid(struct process* p)`

  Returns the `pid_t` of the process p. The `pid_t` type is the same type as `tid_t`, which is `typedef`'d to be an `int`.

Stacks and Limits Each user thread must have its own stack in userspace to do scratch work in. When creating a thread, you must create a stack for it in userspace. See III Page Tables for information on how to do this.

In `userprog/process.h`, we provide the following limits for pthreads:

```
// At most 8MB can be allocated to the stack
#define MAX_STACK_PAGES (1 << 11)
...
#define MAX_THREADS 127
```

In other words, you can only use the top 8MB of user virtual addresses for stacks, and you only need to support at most 127 threads at one time in the system.

3.4 Pthread Library

3.4.1 Threading

A subset of the pthread (Pintos thread) library is provided for you in `lib/user/pthread.h`. These function serve as the glue between the high-level API of `pthread_create`, `pthread_exit`, and `pthread_join` and the low-level system call implementation of these functions. We'll walk you through how the pthread library works by starting at the high-level usage in one of our tests, and walk down the stack until we get to the kernel syscall interface.

- `tests/userprog/multithreading/create-simple.c` In the `create-simple` test, we see how the high-level API of the threading library is supposed to work. The main thread of the process first runs `test_main`. It then creates a new thread to run `thread_function` with the `pthread_check_create` call, and waits for that thread to finish with the `pthread_check_join`. The expected output of this test is shown in `tests/userprog/multithreading/create-simple.ck`.

- The functions `pthread_check_create` and `pthread_check_join` are simple wrappers (defined in `tests/lib.c`) around the "real" functions, `pthread_create` and `pthread_join`, that take in roughly the same values and return the same values as `pthread_create` and `pthread_join`, and ensure that `pthread_create` and `pthread_join` did not fail. The APIs for `pthread_create` and `pthread_join` are:

  `tid_t pthread_create(pthread_fun fun, void* arg)`

  A `pthread_fun` is simply a pointer to a function that takes in an arbitrary `void*` argument, and returns nothing. This is defined in `user/lib/pthread.h`. So, the arguments to `pthread_create` are a function to run, as well as an argument to give that function.

  This function creates a new child thread to run the `pthread_fun` with argument `arg`. This function returns to the parent thread the TID of the child thread, or `TID_ERROR` if the thread could not be created successfully.

  `bool pthread_join(tid_t tid)`

  The caller of this function waits until the thread with TID *tid* finishes executing. This function returns true if *tid* was valid.

- The implementation of `pthread_create` and `pthread_join` are in the file `lib/user/pthread.c`. They each are simple wrappers around the functions `sys_pthread_create` and `sys_pthread_join`, which are syscalls for the OS, that you will be required to implement. Their APIs are similar to `pthread_create` and `pthread_join`, and are as follows:

  `tid_t sys_pthread_create(stub_fun sfun, pthread_fun tfun, const void* arg)`

The sys_pthread_create function creates a new thread to run stub_fun sfun, and gives it as arguments a pthread_fun and a void* pointer, which is intended to be the argument of the pthread_fun. It returns to the parent the TID of the created thread, or TID_ERROR if the thread could not be created.

What is a stub function? There is only one stub function that we are concerned with here, called _pthread_start_stub defined in lib/user/pthread.c, and it's implementation is copied below. This function returns nothing but takes two arguments: a function to run, and an argument for that function. The stub function runs the function on the argument, then calls pthread_exit(). pthread_exit() is a system call that simply kills the current user thread.

```
/* OS jumps to this function when a new thread is created.
   OS is required to setup the stack for this function and
   set %eip to point to the start of this function */
void _pthread_start_stub(pthread_fun fun, void* arg) {
  (*fun)(arg);    // Invoke the thread function
  pthread_exit(); // Call pthread_exit
```

Why this extra layer of indirection? You might have noticed in tests/userprog/multithreading/create-simple.c that pthread_exit() was never called; instead, as soon as the created thread returns from thread_function, it is presumed to have been killed. The stub function is how this is implemented: the OS actually jumps to _pthread_start_stub instead of directly jumping to thread_function when the new thread is created. The stub function then calls thread_function. Then, when thread_function returns, it returns back into _pthread_start_stub. Then, the implementation of pthread_start_stub kills the thread by calling pthread_exit().

`tid_t sys_pthread_join(tid_t tid)`

The caller of this function waits until the thread with TID *tid* finishes executing. This function returns the TID of the child it waited on, or TID_ERROR if it was invalid to wait on that child. See 3.7 User Threads for more information on validity.

`void sys_pthread_exit(void) NO_RETURN`

This function terminates the calling thread. The function pthread_exit simply calls this function.

The functions sys_pthread_create, sys_pthread_join, and sys_pthread_exit are system calls that you are required to implement for this project. They have slightly different APIs than the high level pthread_create, pthread_join, and pthread_exit functions defined in lib/user/pthread.h, but are fundamentally very similar. We have setup the user-side of the syscall interface for you in lib/syscall-nr.h, lib/user/syscall.c, and lib/user/syscall.h, and it is your job to implement these system calls in userprog/ in the kernel. See 3.7 User Threads for more information.

3.4.2 User-Level Synchronization

Our pthread library also provides an interface to user-level synchronization primitives. See lib/user/syscall.h. We define the primitives lock_t and sema_t to represent locks and semaphores in user programs. You can change these definitions if you'd like, but we found the current definitions sufficient for our implementation. We provide the following syscall stubs:

- `bool lock_init(lock_t* lock)`

  Initializes *lock* by registering it with the kernel, and returns true if the initialization was successful. In tests/lib.c, you will see we define lock_check_init, which is analogous to pthread_check_create and pthread_check_join; it simply verifies that the initialization was successful.

- `void lock_acquire(lock_t* lock)`

  Acquires *lock*, and exits the process if acquisition failed. The syscall implementation of `lock_acquire` should return a boolean as to whether acquisition failed; the user level implementation of `lock_acquire` in `lib/user/syscall.c` handles termination of the process. You should **not** update the `lock_acquire` (or for that matter, any of the below functions) code in `lib/user/syscall.c` to remove the `exit` call – it will simply make debugging more difficult.

- `void lock_release(lock_t* lock)`

  Acquires *lock*, and exits the process if the release failed. The syscall implementation of `lock_release` should return a boolean as to whether release failed.

- `bool sema_init(sema_t* sema, int val)`

  Initializes *sema* to *val* by registering it with the kernel, and returns true if the initialization was successful. In `tests/lib.c`, you will see we define `lock_check_init`, which is analogous to `pthread_check_create` and `pthread_check_join`; it simply verifies that the initialization was successful.

- `void sema_down(sema_t* sema)`

  Downs *sema*, and exits the process if the down operation failed. The syscall implementation of `sema_down` should return a boolean as to whether the down operation failed.

- `void sema_up(sema_t* sema)`

  Ups *sema*, and exits the process if the up operation failed. The syscall implementation of `sema_up` should return a boolean as to whether the up operation failed.

Your task will be to implement those system calls in the kernel. On every synchronization system call, you are allowed to make a kernel crossing. In other words, you do not need to avoid kernel crossings like is done in the implementation of `futex`.

Given user-level locks and semaphores, it's possible to implement user-level condition variables entirely at user-level with locks and semaphores as primitives. Feel free to implement condition variables if you would like, but it is not required as part of the project. The implementation will look similar to the implementation of CVs in `threads/synch.c`.

3.5 Efficient Alarm Clock

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

1. If `timer_sleep()` is called with a zero or negative argument, then it should simply return immediately.

2. There exist already-implemented functions `timer_msleep()`, `timer_usleep()`, and `timer_nsleep()` which take the sleep duration argument in units of milliseconds, microseconds, or nanoseconds respectively, which will work once you have implemented `timer_sleep()`. You do not need to modify them further.

3. When Pintos starts up, the clock does **not** run in realtime by default. As such, 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.

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).

3.6 Schedulers

3.6.1 General Information

The skeleton code for Pintos includes a framework to support multiple simultaneous scheduler implementations, of which one will be in use at any time. The choice of which scheduler to execute is controlled by the variable `active_sched_policy`, declared within `threads/thread.c`. Its type is that of an enum with four values by default: `SCHED_FIFO`, `SCHED_PRIO`, `SCHED_FAIR`, and `SCHED_MLFQS`, along with a macro `SCHED_DEFAULT` which resolves to `SCHED_FIFO` by default. You can make Pintos use a specific scheduler by passing in an argument when it starts: for example, `-sched-fair` will start Pintos with the fair scheduler enabled. It is possible, but not recommended, to change the value of `sched_policy` at runtime; similarly, you may change the way in which the scheduler is dispatched to some other method (e.g. a macro) if you wish, as long as your method supports the same kernel flags as the skeleton — this will not be necessary, however, and is not recommended. As long as you do not change this value, you can assume that it does not change at runtime: this is important because it means you don't have to simultaneously maintain, for example, a FIFO thread queue and a priority-based structure, as you would if the scheduling policy could change during runtime.

Of the available scheduling modes, one (`SCHED_FIFO`) is already implemented for you; in addition, you will be implementing support for `SCHED_PRIO` and `SCHED_FAIR`. You do not need to make any changes to support `SCHED_MLFQS`; implementing an MLFQS was at one point part of this project, but that is not presently the case. When you are ready to test your schedulers, you do *not* need to change `SCHED_DEFAULT` to point to `SCHED_PRIO` or `SCHED_FAIR`; the testing framework will specify which scheduler it wants to use for each test.

Some additional points of note:

1. Both `SCHED_PRIO` and `SCHED_FAIR` should be implemented as priority schedulers, meaning that each thread is assigned a priority which changes how much time it gets to spend on the CPU relative to other threads.

2. By default, threads have a priority in the range `PRI_MIN` (0) to `PRI_MAX` (63). You may expand this range if you wish, but you may *not* shrink it: all integers from 0 to 63, inclusive, must be valid priorities in your implementation.

3. 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.

4. Along the same lines as the note in Efficient Alarm Clock, any code running with interrupts disabled should as fast as possible; be careful not to add too much computation to any sections of the scheduler which run with them off.

5. Try to think simple when you're implementing your schedulers: real world schedulers are often surprisingly elegant at their core, and even those are too complicated for this task — we do not expect you to implement anything like Linux's Completely Fair Scheduler, to be clear.

3.6.2 Priority Scheduler

The actual priority scheduler does not require much complexity in and of itself; consider how extant operating systems implement this sort of scheduler if you're confused as to how to approach this in an efficient way.

1. 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.

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

3. 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.

The priority donation component of this task will likely require some thought — it may be helpful to sketch out some scenarios on paper or on a whiteboard to see if your proposed system holds up.

1. 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 whenever a resource is released or a monitor is signaled.

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

3. 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.

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.

3.6.3 Fair Scheduler

The goal of a general-purpose scheduler is to balance threads' different scheduling needs. Users often will want to prioritize certain tasks over others, but they do not want this prioritization to result in lower-priority tasks' starvation.
As such, your design for the fair scheduler must maintain two properties:

I. **Prioritization**: It must give *high*-priority threads proportionally more time on the CPU

II. **Fairness**: It must consistently prevent *low*-priority threads from starving as a result

Clearly, (I) is achieved by the Strict Priority Scheduler, while (II) can be perfectly achieved by, for example, a round-robin scheduler: your scheduler, however, must fulfill both. In a general sense the two goals are in opposition to one another, and as such there exists a broad range of possible designs that could fulfill this task, balancing at different places between those two extremes. The exact bounds on your design are specified in the tests, but broadly, any system which (probabilistically) gives higher-priority threads more time on the CPU than lower-priority ones while still allowing all threads to make forward progress will satisfy this component.

Your solutions do not need to be particularly complex: there are several surprisingly simple approaches which will fulfill all the requirements laid out above.

Beyond the core information above, a few additional helpful points are included below:

- The fair scheduler should use the exact same priority-management methods as the priority scheduler; threads should be able to call `thread_set_priority()` regardless of whether the underlying scheduler mode is `SCHED_PRIO` or `SCHED_FAIR`.
- It is not necessary for `SCHED_FAIR` to include support for priority donation, but it won't cause problems if it does — you do not need to disable it, though you are free to do so.
- Note that "more access" does not necessarily mean *preferential* access: as long as, with a sufficiently big time sample, higher priority threads get *more time* on the CPU than their lower-priority counterparts, requirement (I) is considered to be fulfilled.
- Given that many approaches to this problem will involve the use of randomization, it is important to note that the tests for this component will use a consistent random seed between successive executions. The file `lib/kernel/random.h` will likely be of use in this regard.
- When many threads of different priorities are executing at once, it might be difficult to observe a difference in the actual time allocated to threads with very close priority levels, e.g. 10 and 11. To ensure fair test results, tests focusing on this aspect of the fair scheduler will space priorities out in intervals of 8; of course, you will still need to be able to support threads with priorities anywhere between 0 and 63 at the very least.
- The fair scheduler does not need to prevent *every* possible cause for starvation; if the system is simply overloaded, there is only so much a scheduling algorithm can do to allocate the limited resources at hand. A good rule of thumb is that if a round-robin scheduler would struggle to meet the demands of a given workload, it's okay for your scheduler to struggle when presented with the same.

3.7 User Threads

For this project, you will need to implement the following new system calls:

System Call: tid_t sys_pthread_create(stub_fun sfun, pthread_fun tfun, const void* arg) Creates a new user thread running stub function *sfun*, with arguments *tfun* and *arg*. Returns TID of created thread, or `TID_ERROR` if allocation failed.

System Call: void sys_pthread_exit(void) NO_RETURN Terminates the calling user thread. If the main thread calls `pthread_exit`, it should join on all currently active threads, and then exit the process.

System Call: tid_t sys_pthread_join(tid_t tid) Suspends the calling thread until the thread with TID *tid* finishes. Returns the TID of the thread waited on, or `TID_ERROR` if the thread could not be joined on. It is only valid to join on threads that are part of the same process and have not yet been joined on. It is valid to join on a thread that *was* part of the same process, but has already terminated – in such cases, the `sys_pthread_join` call should not block. Any thread can join on any other thread (the main thread included). If a thread joins on main, it should be woken up and allowed to run after main calls `pthread_exit` but before the process is killed (see above).

System Call: bool lock_init(lock_t* lock) Initializes *lock*, where *lock* is a pointer to a `lock_t` in userspace. Returns true if in initialization was successful.

System Call: bool lock_acquire(lock_t* lock) Acquires *lock*, blocking if necessary, where *lock* is a pointer to a `lock_t` in userspace. Returns true if the lock was successfully acquired, false if the lock was not registered with the kernel in a `lock_init` call or if the current thread already holds the lock.

System Call: bool lock_release(lock_t* lock) Releases *lock*, where *lock* is a pointer to a `lock_t` in userspace. Returns true if the lock was successfully released, false if the lock was not registered with the kernel in a `lock_init` call or if the current thread does not hold the lock.

System Call: bool sema_init(sema_t* sema, int val) Initializes *sema* to *val*, where *sema* is a pointer to a `sema_t` in userspace. Returns true if in initialization was successful.

System Call: bool sema_down(sema_t* sema) Downs *sema*, blocking if necessary, where *sema* is a pointer to a `sema_t` in userspace. Returns true if the semaphore was successfully downed, false if the semaphore was not registered with the kernel in a `sema_init` call.

System Call: bool sema_up(sema_t* sema) Ups *sema*, where *sema* is a pointer to a `sema_t` in userspace. Returns true if the sema was successfully up'd, false if the sema was not registered with the kernel in a `sema_init` call.

System Call: int sys_getschedpolicy() Returns an integer corresponding to the current scheduling policy in use by the kernel.

The definitions of `pid_t`, `stub_fun`, and `pthread_fun` in the kernel are in `userprog/process.h`, and mimic the userspace definitions described in the Reference section.

You will also need to update the system calls you implemented in Project 1 to support multiple user threads. Most of the changes you'll make are short and straightforward, but substantial changes will be made to the process control syscalls. The expected behavior of process control syscalls with respect to multithreaded user programs is outlined below:

- `pid_t exec(const char* file)`

  When either a single-threaded or multithreaded program exec's a new process, the new process should only have a single thread of control, the main thread. New threads of control can be created in the child process with the pthread syscalls.

- `int wait(pid_t)`

  When a user thread waits on a child process, only the user thread that called `wait` should be suspeneded; the other threads in the parent process should be able to continue working.

- `void exit(int status)`

  When exit is called on a multithreaded program, all currently active threads in the user program should be immediately terminated: none of the user threads should be able to execute any more user-level code. Each of the backing kernel threads should release all its resources before terminating.

  We recommend you implement this functionality *without* keeping a list of all resources a kernel thread can have. As a hint and simplifying assumption, you may assume that a user thread that enters the kernel never blocks indefinitely. You are not required to make use of this assumption, but it will make implementation of this section much easier.

  The assumption above is not true in a number of scenarios, which our test suite simply ignores. For clarity, we list a few such scenarios: (1) a user thread calls `wait` on a child process that infinite loops, (2) two user threads deadlock with their own user-level synchronization primitives, or (3) a user thread is waiting on STDIN, which may never arrive.

  The assumption above does *not* apply to the case where threads are waiting on other threads in the same process through `pthread_join`. Joiners should still be woken up with the thread they joined on is killed, and joiners on the exiting thread should also be woken up.

3.7.1 Synchronization

To ease implementation difficulty, we will *not* be requiring you to implement fine-grained synchronization for syscalls within multithreaded programs. You are allowed to serialize actions per-process (but not globally).

3.7.2 Additional Information

- **Exit Codes:** (1) If the main thread calls `pthread_exit`, the process should terminate with exit code 0. (2) If any thread calls `exit(n)`, the process should terminate with exit code n. (3) If the process terminates with an exception, it should exit with exit code -1. These are listed in priority order (with 3 being the highest priority), in the sense that if any of these occur simultaneously, the exit code should be the exit code corresponding termination with the highest prioirty. For example, if main calls `pthread_exit` and while it is waiting for user threads to finish, one of them terminates with an exception, the exit code should be set to -1. Also, if multiple calls to `exit(n)` are made at the same time with different values of `n`, any choice of `n` is valid. Treat exit code rules as secondary: we will not test you on them in design review, and you should only be concerned about them if you are failing a test because of the wrong exit code.

- Switching between user threads and switching between user processes require different actions on part of the kernel. Specifically, for switches between processes, (1) the page table base pointer must be updated and (2) any virtual caches (which for our purposes, is the TLB) should be invalidated. For switches between user threads, both of these things should be avoided. This is already done for you in `process_activate`, which is called everytime a new thread is created in `load` and everytime a new thread scheduled in `thread_schedule_tail`. Don't forget to activate the process when you create a new user thread.

- You are not required to augment the scheduler for Task 3: User Threads; you can just let the scheduler treat all threads the same, even if they belong to the same process. As a pathological example, if a user program A has 100 threads, and a user program B has only 1 thread, most of

the CPU will be dominated by A's threads, and B's thread will be starved. You are **not** required to augment the scheduler to make this scenario more fair.

- Task 3: User Threads should be able to be implemented independently of Task 1: Efficient Alarm Clock and Task 2: Schedulers. The alarm clock does not have an exposed interface via system calls, so Task 1: Efficient Alarm Clock and Task 3: User Threads are completely independent. There is some overlap between Task 2: Schedulers and Task 3: User Threads, because Task 3: User Threads uses both the scheduler and locks. However, the tasks should still be fairly independent of one another, since all user threads should have the same priority (`PRI_DEFAULT`).

- As our test programs are multithreaded, the `console_lock` defined in `tests/lib.c` is essential; threads can acquire this during printing calls to make sure print output of different threads is not interleaved. Currently, the test code only uses the console lock when `syn_msg` (defined in `tests/lib.c`) is set to true. The console lock is initialized in `tests/main.c` before `test_main` is called in each of the tests. Because the console lock is a user-level lock, it will only work after you have implemented user-level locking. Until you've implemented user-level locking, **all your tests will fail** as a result of console lock initialization; you can comment out the line `lock_init(&console_lock)` in `tests/main.c` to temporarily prevent this issue.

- In `threads/interrupt.c`, you will find the function

  ```
  static inline bool is_trap_from_userspace(struct intr_frame* frame)
  ```

  which is written return true if this interrupt represents a transition from user-mode to kernel-mode. You might find this helpful for this project.

- Workflow Recommendations: this task is most easily done in small steps. Start by implementing a barebones `pthread_create` and `pthread_execute` so that
  you pass `tests/userprog/multithreading/ create-simple`. Then, slowly add more and more features. It is easier to augment a working design than to fix a broken one. Make sure to **carefully track resources**. Everything that you allocate *must be freed*!

4 Appendix

I 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.

I.a 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. 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.

I.b Semaphores

A **semaphore** is a non-negative 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`.

I.c 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.

I.d 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.

I.e Readers-Writers Locks

In the starter code, we provide you with a working implementation of a readers-writers lock, should you choose to use it. The API for the lock is defined in `threads/synch.h`, and is shown below for convenience.

```
/* Readers-writers lock. */
#define RW_READER 1
#define RW_WRITER 0

struct rw_lock {
  struct lock lock;
```

```
  struct condition read, write;
  int AR, WR, AW, WW;
};

void rw_lock_init(struct rw_lock*);
void rw_lock_acquire(struct rw_lock*, bool reader);
void rw_lock_release(struct rw_lock*, bool reader);
```

You can initialize a readers-writers lock with the `rw_lock_init` function, which mirrors the standard `lock_init` function. Unlike a standard lock, you may acquire a readers-writers lock in either reader mode or writer mode, which is specified by `bool reader`.. You can use the definitions of `RW_READER` and `RW_WRITER` for cleaner code. One example usage (for a readers lock on the stack) is shown below:

```
void demo_rw_lock_function() {
    // Allocate bytes for rw_lock on the stack
    struct rw_lock my_rw_lock;

    // Initialize the RW lock
    rw_lock_init(&my_rw_lock);

    // Acquire RW lock in reader mode
    rw_lock_acquire(&rw_lock, RW_READER);

    // Release RW lock in reader mode
    rw_lock_release(&rw_lock, RW_READER);
}
```

I.f 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.

II 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.

II.a 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 very relevant in this project, since some of the threads you will be dealing with are kernel threads and some of the threads you will be dealing with are user threads.

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`.

II.b 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).

III Page Tables

The code in pagedir.c is an abstract interface to the 80x86 hardware page table, also called a "page directory" by Intel processor documentation. The page table interface uses a `uint32_t *` to represent a page table because this is convenient for accessing their internal structure. The sections below describe the page table interface and internals.

III.a Creation, Destruction, and Activation

These functions create, destroy, and activate page tables. The base Pintos code already calls these functions where necessary, so it should not be necessary to call them yourself.

- **Function** `uint32_t *pagedir_create (void)`

  Creates and returns a new page table. The new page table contains Pintos's normal kernel virtual page mappings, but no user virtual mappings.

  Returns a null pointer if memory cannot be obtained.

- **Function** `void pagedir_destroy (uint32_t *pd)`

  Frees all of the resources held by *pd*, including the page table itself and the frames that it maps.

- **Function** `void pagedir_activate (uint32_t *pd)`

  Activates *pd*. The active page table is the one used by the CPU to translate memory references.

III.b Inspection and Updates

These functions examine or update the mappings from pages to frames encapsulated by a page table. They work on both active and inactive page tables (that is, those for running and suspended processes), flushing the TLB as necessary.

- **Function** `bool pagedir_set_page (uint32_t *pd, void *upage, void *kpage, bool writable)`

  Adds to *pd* a mapping from user page *upage* to the frame identified by kernel virtual address *kpage*. If *writable* is true, the page is mapped read/write; otherwise, it is mapped read-only.

  User page *upage* must not already be mapped in pd.

  Kernel page *kpage* should be a kernel virtual address obtained from the user pool with `palloc_get_page(PAL_USER)`.

  Returns true if successful, false on failure. Failure will occur if additional memory required for the page table cannot be obtained.

- **Function** `void *pagedir_get_page (uint32_t *pd, const void *uaddr)`

  Looks up the frame mapped to *uaddr* in *pd*. Returns the kernel virtual address for that frame, if *uaddr* is mapped, or a null pointer if it is not.

- **Function** `void pagedir_clear_page (uint32_t *pd, void *page)`

  Marks page "not present" in *pd*. Later accesses to the page will fault.

  Other bits in the page table for *page* are preserved.

  This function has no effect if *page* is not mapped.

III.c Page Table Details

The functions provided with Pintos are sufficient to implement the projects. However, you may still find it worthwhile to understand the hardware page table format, so we'll go into a little detail in this section.

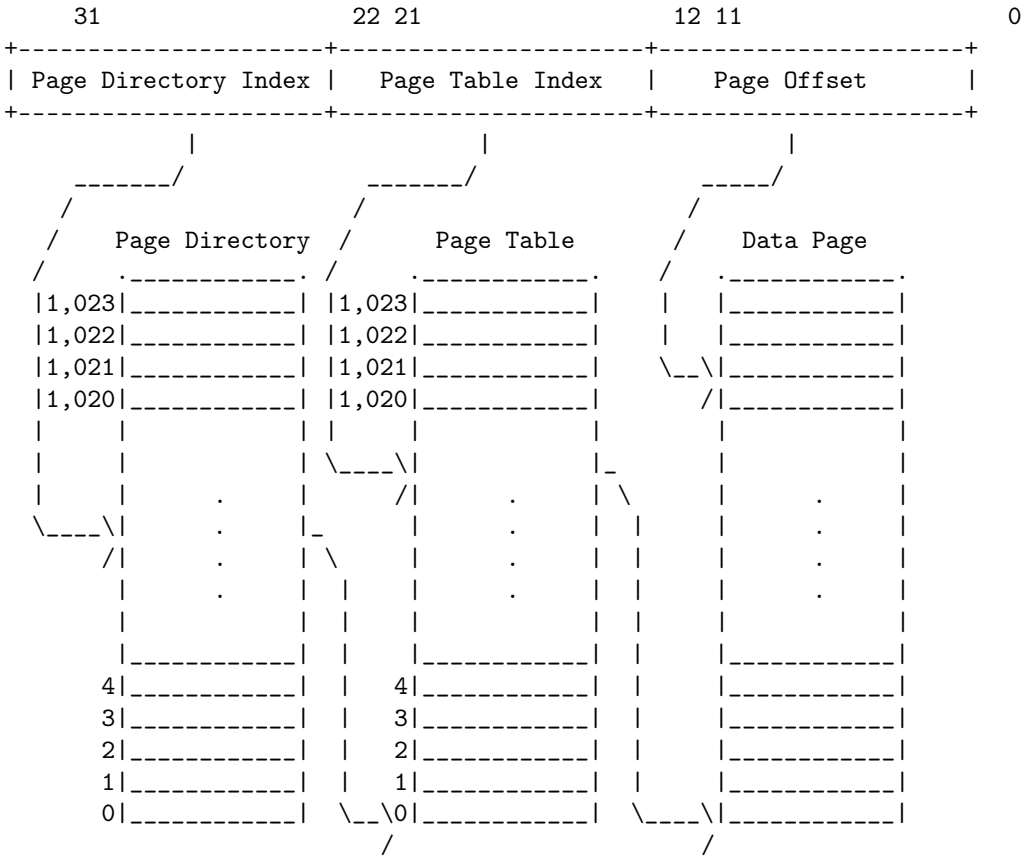
The top-level paging data structure is a page called the "page directory" (PD) arranged as an array of 1,024 32-bit page directory entries (PDEs), each of which represents 4 MB of virtual memory. Each PDE may point to the physical address of another page called a "page table" (PT) arranged, similarly, as an array of 1,024 32-bit page table entries (PTEs), each of which translates a single 4 kB virtual page to a physical page.

Translation of a virtual address into a physical address follows the three-step process illustrated in the diagram below:

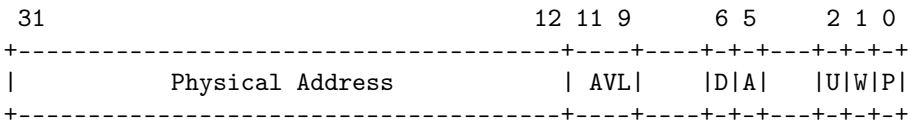
1. The most-significant 10 bits of the virtual address (bits 22...31) index the page directory. If the PDE is marked "present," the physical address of a page table is read from the PDE thus obtained. If the PDE is marked "not present" then a page fault occurs.

2. The next 10 bits of the virtual address (bits 12...21) index the page table. If the PTE is marked "present," the physical address of a data page is read from the PTE thus obtained. If the PTE is marked "not present" then a page fault occurs.

3. The least-significant 12 bits of the virtual address (bits 0...11) are added to the data page's physical base address, yielding the final physical address.



For the most part, you do not need to understand the PTE format to do the Pintos projects. The actual format of a page table entry is summarized below. For complete information, refer to section 3.7, "Page Translation Using 32-Bit Physical Addressing," in IA32-v3a[3].



Pintos provides some macros and functions that are useful for working with raw page tables. The most helpful ones, found in `threads/pte.h` are:

- **Macro** `PTE_P`

  Bit 0, the "present" bit. When this bit is 1, the other bits are interpreted as described below. When this bit is 0, any attempt to access the page will page fault. The remaining bits are then not used by the CPU and may be used by the OS for any purpose.

- **Macro** `PTE_ADDR`

  Bits 12...31, the top 20 bits of the physical address of a frame. The low 12 bits of the frame's address are always 0.

[3] https://web.stanford.edu/class/cs140/projects/pintos/pintos_13.html#IA32-v3a

IV 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).

V 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.

The following example illustrates how one might debug a project 1 solution in which occasionally a thread that calls `timer_sleep` is not woken up; thanks to Godmar Black for providing the sample debugging session. With this bug, tests such as `mlfqs_load_1` get stuck. This semester, we will not be implementing the Multi-Level Feedback Queue Scheduler (MLFQS), but the example should still be valuable. The source code for these tests are still available in `tests/threads/` if you want to follow along. If you want to run MLFQS tests, you can do so by updating `tests/threads/Make.tests`.

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.

Sample GDB Session .

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.