Project 2: Threads

Due: April 2, 2022

Contents

1 Introduction **3**
1.1 Setup . . . 3

2 Tasks **4**
2.1 Efficient Alarm Clock . . . 4
2.2 Strict Priority Scheduler . . . 4
2.3 User Threads . . . 4
2.4 Concept Check . . . 5
2.5 Testing . . . 6

3 Deliverables **7**
3.1 Design . . . 7
3.1.1 Document . . . 7
3.1.2 Review . . . 8
3.1.3 Grading . . . 8
3.2 Code . . . 8
3.2.1 Checkpoints . . . 8
3.2.2 Testing . . . 8
3.2.3 Quality . . . 8
3.3 Report . . . 9
3.4 Evaluations . . . 9
3.5 Submission . . . 10
3.6 Grading . . . 10

4 Plan **11**
4.1 Checkpoint 1 . . . 11
4.2 Checkpoint 2 . . . 11
4.3 Final . . . 11

5 FAQ **12**
5.1 Efficient Alarm Clock . . . 13
5.2 Strict Priority Scheduler . . . 13

A Threads **14**
A.1 Understanding Threads . . . 14
A.2 The Thread Struct . . . 14
A.3 Thread Functions . . . 16

B Scheduler **17**

C User Threads **18**

C.1 Implementation Requirements . . . 18
C.2 Synchronization . . . 19
C.3 Additional Information . . . 19

D Pthread Library **21**
D.1 Threading . . . 21
D.2 User-Level Synchronization . . . 22

E Synchronization **24**
E.1 Disabling Interrupts . . . 24
E.2 Semaphores . . . 25
E.3 Locks . . . 25
E.4 Monitors . . . 26
E.5 Optimization Barriers . . . 27

F Advice **29**
F.1 Group Work . . . 29
F.1.1 Meetings . . . 29
F.2 Development . . . 29
F.2.1 Compiler Warnings . . . 29
F.2.2 Faster Compilation . . . 30
F.2.3 Repeated Commands . . . 30
F.2.4 Hail Mary . . . 30

1 Introduction

Welcome to Project Threads! In this project, you will add features to the threading system of Pintos.

In Project User Programs, 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 was the *only* thread in that process; multithreaded user programs were not supported. This project, you will be overcoming this limitation by adding support for multithreaded user programs. Moreover, you will be implementing an efficient alarm clock and strict priority scheduler. However, for simplicity, you will implement these features only for kernel threads – threads that only execute in the kernel mode and have no userspace component.

1.1 Setup

First, log into your VM. You should already have your Pintos code from Project User Programs, which you will be building off of. We recommend that you tag your final Project User Programs code since you will probably build off of it for Project File Systems.

```
> cd ~/code/group
> git tag proj-userprog-completed
> git push group master --tags
```

If you are using the staff solution for Project User Programs (check Piazza for more information), you will need to reset your repo.

```
> git fetch staff master
> rm -rf src/
> git checkout staff/master -- src
> git commit -m "Reset to skeleton code before applying diff"
> git pull staff master
> git push group master
```

Then, you can apply the given patch `p1.diff`. Make sure `p1.diff` is in `/code/group`.

```
> patch -p4 -i p1.diff
```

For this project, `group0` has been updated to include important functions to implement for multi-threading. Please merge the starter code before starting Project 2 using `git pull staff master`. Because of this update, you will likely get merge conflicts in `process.c` and `process.h`. Please resolve the merge conflicts. [This guide[1] may be helpful. It will also be helpful to take a look at what is being updated.]

[1] https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-using-the-command-line

2 Tasks

2.1 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 reimplement `timer_sleep` so that it executes efficiently without any busy waiting.

2.2 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` and get its own priority by calling `thread_get_prioirty`.

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.

2.3 User Threads

Pintos is a multithreaded kernel (i.e. there can be more than one thread running in the kernel). While working on Project User Programs, you have no doubt worked with the threading interface in the kernel. In `threads/thread.c`, `thread_create` allows us to create a new kernel thread that runs a specific kernel task. Analogously, 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 User Programs, 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 POSIX `pthread` library.

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

In Project User Programs, 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 this task, 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. 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.

2.4 Concept Check

The following are conceptual questions meant to be answered in your design document. You won't need to write any code for these questions, but you may need to read and reference some.

1. When a kernel thread in Pintos calls `thread_exit`, when and 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. However, 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?

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

5. Consider a fully-functional and 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.**

2.5 Testing

Pintos already contains a test suite for Project User Programs, but not all of the parts of this project have complete test coverage. You must submit **one new test case** which exercise functionality that is not covered by existing tests. We will not tell you what features to write tests for, so you will be responsible for identifying which features of this project would benefit most from additional tests. Make sure your own project implementation passes the tests that you write. You can pick any appropriate name for your test, but beware that **test names should be no longer than 14 characters**.

3 Deliverables

3.1 Design

Before you start writing any code for your project, you need to create an design plan for each feature and convince yourself that your design is correct. You must **submit a design document** and **attend a design review** with your TA. This will help you solidify your understanding of the project and have a solid attack plan before tackling a large codebase.

3.1.1 Document

Like any techinical writing, your design document needs to be clean and well formatted. We've provided you with a template linked on the website that you must use. The template can be found on the website. We use Dropbox Paper[3] which supports real-time collaboration like Google Docs with the added benefit of techincal writing support (e.g. code blocks, LaTeX). **Not using this template or failure to use code formatting will result in losing points**. The main goal of this is not to punish you for this but rather make it easy for TAs to read.

To get started, navigate to the template and click the "Create doc" button on the top right hand corner. You can share this doc with other group members to collaborate on it. **Make sure to click the blue "Share" button in *your* document, not the template**.

For each task except for Concept Check, you must explain the following aspects of your proposed design. We suggest you create a section for each task which has subsections for each of the following aspects.

Data Structures and Functions

List any `struct` definitions, global or static variables, `typedef`s, or enumerations that you will be adding or modifying (if it already exists). These should be **written with C not pseudocode**. Include a **brief explanation** (i.e. a one line comment) of the purpose of each modification. A more in depth explanation should be left for the following sections.

Algorithms

Tell us how you plan on writing the necessary code. Your description should be at a level below the high level description of requirements given in the assignment. **Do not repeat anything that is already stated on the spec**.

On the other hand, your description should be at a level above the code itself. Don't give a line-by-line rundown of what code you plan to write. You may use *small snippets* of pseudocode or C in places you deem appropriate. Instead, you need to convince us that your design satisfies all the requirements, **especially any edge cases**. We expect you to have read through the Pintos source code when preparing your design document, and your design document should refer to the appropriate parts of the Pintos source code when necessary to clarify your implementation.

Synchronization

List all resources that are shared across threads and processes. For each resource, explain how the it is accessed (e.g. from an interrupt context) and describe your strategy to ensure it is shared and modified safely (i.e. no race conditions, deadlocks).

In general, the best synchronization strategies are simple and easily verifiable. If your synchronization strategy is difficult to explain, this is a good indication that you should simplify your strategy. Discuss the time and memory costs of your synchronization approach, and whether your strategy will significantly limit the concurrency of the kernel and/or user processes/threads. When discussing the concurrency allowed by your approach, explain how frequently threads will contend on the shared resources, and any limits on the number of threads that can enter independent critical sections at a single time. You should aim to avoid locking strategies that are overly coarse.

[3] `http://paper.dropbox.com/`

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.

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.

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.

3.1.2 Review

After you submit your design doc, you will schedule a design review with your TA. A calendar signup link will be posted sometime before the design doc due date. 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 participation points.

3.1.3 Grading

The design document and design review will be graded together. Your score will reflect how convincing your design is, based on your explanation in your design document and your answers during the design review. If you cannot make a design review, please contact your TA to work out an arrangement. **An unexcused absence from a design review will result in a 0 for the design portion**.

3.2 Code

Code will be submitted to GitHub via your groupX repo. Pintos comes with a test suite that you can run locally on your VM. We will run the same tests on the autograder, meaning there are **no hidden tests**. As a result, we recommend you test locally as much as possible since the autograder's bandwidth is limited.

3.2.1 Checkpoints

On the autograder, we will provide checkpoints for you to stay on pace with the project. **Checkpoints will not be graded and have no effect on your grade**. However, we still encourage students to keep up with the checkpoints See Plan for more details on the specific checkpoints for this project.

Each checkpoint will have a corresponding autograder. Checkpoint 1 autograder will automatically run until its deadline, then Checkpoint 2 will automatically run until its deadline. The final autograder will not automatically run until after the Checkpoint 2 autograder. If you'd like, you can manually trigger the autograders as they'll be available throughout the entire project duration.

3.2.2 Testing

Your testing code needs to be included in your repo as well under the appropriate folder.

3.2.3 Quality

The score of your code will be mainly determined by your autograder score. However, you will also be graded on the quality of your code on some factors including but not limited to

- Does your code exhibit any major memory safety problems (especially regarding strings), memory leaks, poor error handling, or race conditions?
- Is your code simple and easy to understand? Does it follow a consistent naming convention?
- Did you add sufficient comments to explain complex portions of code?
- Did you leave commented-out code in your final submission?
- Did you copy and paste code instead of creating reusable functions?
- Did you re-implement linked list algorithms instead of using the provided list manipulation functions?
- Is your Git commit history full of binary files?

Note that you don't have to worry about manually enforcing code formatting (e.g. indentation, spacing, wrapping long lines, consistent placement of braces.) as long as you setup your precommit hook correctly in Project User Programs. You may also find it helpful to format code every once in a while by running `make format`.

3.3 Report

After you complete the code for your project, your group will write a report reflecting on the project. While you're not expected to write a massive report, we are asking for the same level of detail as expected for the design document. Your report should include the following sections.

Changes

Discuss any changes you made since your initial design document. Explain why you made those changes. Feel free to reiterate what you discussed with your TA during the design review if necessary.

Reflection

Discuss the contribution of each member. Make sure to be specific about the parts of each task each member worked on. Reflect on the overall working environment and discuss what went well and areas of improvement.

Testing

For each of the 2 test cases you write, provide

- Description of the feature your test case is supposed to test.
- Overview of how the mechanics of your test case work, as well as a *qualitative* description of the expected output.
- Output and results of your own Pintos kernel when you run the test case. These files will have the extensions `.output` and `.result`.
- Two non-trivial potential kernel bugs and how they would have affected the output of this test case. Express these in the form "If my kernel did X instead of Y, then the test case would output Z instead.". You should identify two different bugs per test case, but you can use the same bug for both of your two test cases. These bugs should be related to your test case (e.g. syntax errors don't

In addition, tell us about your experience writing tests for Pintos. What can be improved about the Pintos testing system? What did you learn from writing test cases?

3.4 Evaluations

After you finish all the components above, you must submit an evaluation of your group members. You are will be given $20(n-1)$ points to distribute amongst your group members *not including yourself*, where n is the number of people in your group. For instance, if you have four team members, you will get 60 points to distribute amongst the rest of your group members. If you believe all group members contributed equally, you would give each member 20 points each.

You will also fill out the details of what each member worked on. While this is similar to Report portion of the report, this will serve as a space to truthfully speak about the contributions since the report is a collaborative document.

To submit evaluations, navigate to your `personal` repo and pull from the staff repo. You should see a folder for evaluations (e.g. `proj-userprog-eval` for Project User Programs) which contains a `evals.txt`. Fill out the `evals.txt` file. Each line should pertain to one group member, so make your comments for each group member remain in that line. The following is an example of what your csv should look like.

```
Name|Autograder ID (XXX)|Score|Comment
Jieun Lee|516|23|Worked on all tasks finished the testing report
Taeyeon Kim|309|20|Worked on tasks 2, 3, 4, helped come up with a lot of design ideas.
Sean Kim|327|17|Helped with task 1 and testing but didn't work on design doc.
```

Your comments should be longer and more descriptive than the ones given above. Ideally, aim to write between 100 and 200 words for each group member.

This evaluation will remain anonymous from the rest of your group members. If we notice some extreme weightings of scores, we will reach out and arrange a meeting to discuss any group issues. **These evaluations are important and hold substantial weight, so fill them out honestly and thoroughly**. See Grading for more information on how evaluations will be used.

3.5 Submission

Design documents and final reports should be submitted to Gradescope to their respective locations. You can export your document from Dropbox Paper by clicking on the three dots in the top right hand corner and clicking “Export”.

Make sure you've pushed your code and have an autograder build. This build **must include the testing code** for you to receive credit on testing.

Make sure to push your evals to your individual repo to trigger the autograder. If you get errors from the autograder, make sure to check that you've done the following.

- Included exactly $n - 1$ members (i.e. no evals of yourself).
- Autograder IDs are correct and *not prefixed with student* (i.e. 162 instead of student162).
- The points sum to exactly $20(n - 1)$.
- No negative points for any member.
- Header row is present.
- Each line contains one evaluation. Your comment section should not consist of multiple lines.
- No extra lines at the end of the file.

Make sure to not change the header row.

3.6 Grading

The components above will be weighed as follows: 15% Design, 70% Code, 15% Report. While evaluations are not explicitly part of your grade, your project score will be impacted based on them. To keep evaluations fair and anonymous, we will not be releasing how evaluations are factored in nor the final scores calculated after factoring in evaluations.

Slip days or extensions cannot be applied on design documents. TAs need enough time to read these design documents to be able to hold design reviews in a timely manner. Slip days will be applied as a maximum across all the other parts. **Evaluation submission will be factored into slip day usage**. For instance, if code is submitted on time, report is submitted one hour late, three members submit their evaluations on time, and one member submits their evaluation two days late, the number of slip days used will be 2 days.

4 Plan

We provide you a suggested order of implementation as well as the specifications for each checkpoint based on our ands past students' experiences. However, this is merely a suggestion and you may elect to approach the project entirely differently.

Keep in mind that checkpoints are not graded.

4.1 Checkpoint 1

Start the project with implementing the efficient alarm clock. This is the simplest task within the project, so try not to spend too much time on it.

4.2 Checkpoint 2

Implement your strict priority scheduler.

4.3 Final

Finish the project by implementing user threads. If you get stuck on `multi-oom` or `exit-clean` tests, we advise that you skip those and come back at the end.

5 FAQ

How much code will I need to write?

Here's a summary of our reference solution. The diffs are from The reference solution represents just one possible solution. Many other solutions are also possible and many of those differ greatly from the reference solution. Some excellent solutions may not modify all the files modified by the reference solution, and some may modify files not modified by the reference solution.

```
devices/timer.c       |   35 +++
threads/interrupt.c   |   19 +
threads/synch.c       |  105 ++++++++++-
threads/synch.h       |    9
threads/thread.c      |  207 ++++++++++++++++++++-
threads/thread.h      |   22 ++
userprog/exception.c  |    2
userprog/process.c    |  485 +++++++++++++++++++++++++++++++++++++++++++++++-----
userprog/process.h    |   67 ++++++-
userprog/syscall.c    |  476 ++++++++++++++++++++++++++++++++++++++++++++++------
userprog/syscall.h    |   10 +
11 files changed, 1309 insertions(+), 128 deletions(-)
```

The kernel always panics when I run a custom test case.

Is your file name too long? The file system limits file names to 14 characters. Is the file system full? Does the file system already contain 16 files? The base Pintos file system has a 16-file limit.

The kernel always panics with `assertion is_thread(t)` failed.

This happens when you overflow your kernel stack. If you're allocating large structures or buffers on the stack, try moving them to static memory or the heap instead. It's also possible that you've made your `struct thread` too large. See the comment underneath the kernel stack diagram in `threads/thread.h` about the importance of keeping your `struct thread` small.

All my user programs die with page faults.

This will happen if you haven't implemented argument passing (or haven't done so correctly). The basic C library for user programs tries to read argc and argv off the stack. If the stack isn't properly set up, this causes a page fault.

All my user programs die upon making a syscall.

You'll have to implement syscall before you see anything else. Every reasonable program tries to make at least one syscall (`exit`) and most programs make more than that. Notably, `printf` invokes the `write` syscall. The default syscall handler just handles `exit()`. Until you have implemented syscalls sufficiently, you can use `hex_dump` to check your argument passing implementation (see **??**).

How can I disassemble user programs?

The `objdump (80x86)` or `i386-elf-objdump` (SPARC) utility can disassemble entire user programs or object files. Invoke it as `objdump -d <file>`. You can use GDB's codedisassemble command to disassemble individual functions.

Why do many C include files not work in Pintos programs? Can I use libfoo in my Pintos programs?

The C library we provide is very limited. It does not include many of the features that are expected of a real operating system's C library. The C library must be built specifically for the operating system (and architecture), since it must make syscalls for I/O and memory allocation. Not all functions do, of course, but usually the library is compiled as a unit.

If the library makes syscalls (e.g, parts of the C standard library), then they almost certainly will not work with Pintos. Pintos does not support as rich a syscall interfaces as real operating systems (e.g., Linux, FreeBSD), and furthermore, uses a different interrupt number (0x30) for syscalls than is used in Linux (0x80).

The chances are good that the library you want uses parts of the C library that Pintos doesn't implement. It will probably take at least some porting effort to make it work under Pintos. Notably, the Pintos user program C library does not have a `malloc` implementation.

How do I compile new user programs?

Modify `examples/Makefile`, then run `make`.

Can I run user programs under a debugger?

Yes, with some limitations. See Project Pregame appendix.

What's the difference between `tid_t` and `pid_t`?

A `tid_t` identifies a kernel thread, which may have a user process running in it (if created with `process_execute`) or not (if created with `thread_create`). It is a data type used only in the kernel.

A `pid_t` identifies a user process. It is used by user processes and the kernel in the `exec` and `wait` syscalls.

You can choose whatever suitable types you like for `tid_t` and `pid_t`. By default, they're both `int`. You can make them a one-to-one mapping, so that the same values in both identify the same process, or you can use a more complex mapping. It's up to you.

5.1 Efficient Alarm Clock

Are we allowed to manually disable and enable interrupts in `timer_sleep`?

Yes. To be precise, interrupts must be on before and after the method call ("after" being before `timer_sleep` returns). It is not acceptable to turn off interrupts during codetimer_sleep and re-enable them at a later time after the function completes.

5.2 Strict Priority Scheduler

What does it mean for synchronization primitives to prefer higher priority threads over lower priority threads?

You should unblock the thread that has the highest priority for any of these synchronization variables. When choosing the next thread to up the semaphore / acquire the lock, you should choose the thread with the highest priority. Similar logic applies when choosing which thread to signal. For broadcasting, you can think about it as signalling until your list of waiters is empty. You should make sure that the highest priority thread gets to the resource first.

A Threads

A.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.

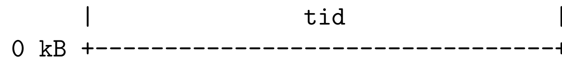
Using GDB, try tracing through a context switch to see what happens. You can set a breakpoint on `schedule()` to start out, and then single-step from there (use "`step`" instead of "`next`"). Be sure to keep track of each thread's address and state, and what procedures are on the call stack for each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

A.2 The Thread Struct

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

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

```
                    4 kB +---------------------------------+
                         |          kernel stack           |
                         |                |                |
                         |                |                |
                         |                V                |
                         |         grows downward          |
                         |                                 |
                         |                                 |
                         |                                 |
                         |                                 |
                         |                                 |
                         |                                 |
sizeof (struct thread) +---------------------------------+
                         |              magic              |
                         |                :                |
                         |                :                |
                         |              status             |
```



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 ?? section for more details.

- **Member of struct thread:** `tid_t tid`
  The thread's thread identifier or *tid*. Every thread must have a tid that is unique over the entire lifetime of the kernel. By default, `tid_t` is a `typedef` for `int` and each new thread receives the numerically next higher tid, starting from 1 for the initial process.
- **Member of struct thread:** `enum thread_status status`
  The thread's state, one of the following:
  - **Thread State:** `THREAD_RUNNING`
    The thread is running. Exactly one thread is running at a given time. `thread_current()` returns the running thread.
  - **Thread State:** `THREAD_READY`
    The thread is ready to run, but it's not running right now. The thread could be selected to run the next time the scheduler is invoked. Ready threads are kept in a doubly linked list called `ready_list`.
  - **Thread State:** `THREAD_BLOCKED`
    The thread is waiting for something, e.g. a lock to become available, an interrupt to be invoked. The thread won't be scheduled again until it transitions to the `THREAD_READY` state with a call to `thread_unblock()`. This is most conveniently done indirectly, using one of the Pintos synchronization primitives that block and unblock threads automatically.
  - **Thread State:** `THREAD_DYING`
    The thread has exited and will be destroyed by the scheduler after switching to the next thread.
- **Member of struct thread:** `char name[16]`
  The thread's name as a string, or at least the first few characters of it.
- **Member of struct thread:** `uint8_t *stack`
  Every thread has its own stack to keep track of its state. When the thread is running, the CPU's stack pointer register tracks the top of the stack and this member is unused. But when the CPU switches to another thread, this member saves the thread's stack pointer. No other members are needed to save the thread's registers, because the other registers that must be saved are saved on the stack.

  When an interrupt occurs, whether in the kernel or a user program, an "`struct intr_frame`" is pushed onto the stack. When the interrupt occurs in a user program, the "`struct intr_frame`" is always at the very top of the page.
- **Member of struct thread:** `int priority`
  A thread priority, ranging from `PRI_MIN` (0) to `PRI_MAX` (63). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. Pintos currently ignores these priorities, but you will implement priority scheduling in this project.
- **Member of struct thread:** `struct list_elem allelem`
  This "list element" is used to link the thread into the list of all threads. Each thread is inserted into this list when it is created and removed when it exits. The `thread_foreach()` function should be used to iterate over all threads.

- **Member of struct thread:** `struct list_elem elem`
  A "list element" used to put the thread into doubly linked lists, either `ready_list` (the list of threads ready to run) or a list of threads waiting on a semaphore in `sema_down()`. It can do double duty because a thread waiting on a semaphore is not ready, and vice versa.
- **Member of struct thread:** `uint32_t *pagedir`
  (Used in Projects Userprog and Filesys.) The page table for the process, if this is a user process.
- **Member of struct thread:** `unsigned magic`
  Always set to `THREAD_MAGIC`, which is just an arbitrary number defined in `threads/thread.c`, and used to detect stack overflow. `thread_current()` checks that the `magic` member of the running thread's `struct thread` is set to `THREAD_MAGIC`. Stack overflow tends to change this value, triggering the assertion. For greatest benefit, as you add members to `struct thread`, leave `magic` at the end.

A.3 Thread Functions

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

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

B 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 mediumpriority 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.

C User Threads

C.1 Implementation Requirements

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: tid_t get_tid(void) Returns the TID of the calling thread.

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

You will also need to update the system calls you implemented in Project User Programs 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.

C.2 Synchronization

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

C.3 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 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.
- User threads should be able to be implemented independently of the efficient alarm clock and strict priority scheduler. The alarm clock does not have an exposed interface via system calls, so the efficient alarm clock and user threads are completely independent. There is some overlap between strict priority

scheduler and user threads because use threads use 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 `is_trap_from_userspace` which will 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 bare-bones `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*!

D Pthread Library

D.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.

`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.

D.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`.

E 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.

E.1 Disabling Interrupts

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

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

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

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

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

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

This project only requires accessing a little bit of thread state from interrupt handlers. For the alarm clock, the timer interrupt needs to wake up sleeping threads. In the advanced scheduler, the timer interrupt needs to access a few global and per-thread variables. When you access these variables from kernel threads, you will need to disable interrupts to prevent the timer interrupt from interfering.

When you do turn off interrupts, take care to do so for the least amount of code possible, or you can end up losing important things such as timer ticks or input events. Turning off interrupts also increases the interrupt handling latency, which can make a machine feel sluggish if taken too far.

The synchronization primitives themselves in synch.c are implemented by disabling interrupts. You may need to increase the amount of code that runs with interrupts disabled here, but you should still try to keep it to a minimum.

Disabling interrupts can be useful for debugging, if you want to make sure that a section of code is not interrupted. You should remove debugging code before turning in your project. (Don't just comment it out, because that can make the code difficult to read.)

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

E.2 Semaphores

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

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

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

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

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

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

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

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

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

E.3 Locks

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

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

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

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

- **Type:** `struct lock`
  Represents a lock.
- **Function:** `void lock_init (struct lock *lock)`
  Initializes `lock` as a new lock. The lock is not initially owned by any thread.
- **Function:** `void lock_acquire (struct lock *lock)`
  Acquires `lock` for the current thread, first waiting for any current owner to release it if necessary.
- **Function:** `bool lock_try_acquire (struct lock *lock)`
  Tries to acquire `lock` for use by the current thread, without waiting. Returns true if successful, false if the lock is already owned. Calling this function in a tight loop is a bad idea because it wastes CPU time, so use `lock_acquire` instead.
- **Function:** `void lock_release (struct lock *lock)`
  Releases `lock`, which the current thread must own.
- **Function:** `bool lock_held_by_current_thread (const struct lock *lock)`
  Returns true if the running thread owns `lock`, false otherwise. There is no function to test whether an arbitrary thread owns a lock, because the answer could change before the caller could act on it.

E.4 Monitors

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

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

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

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

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

  Sending a signal and waking up from a wait are not an atomic operation. Thus, typically `cond_wait`'s caller must recheck the condition after the wait completes and, if necessary, wait again.
- **Function:** `void cond_signal (struct condition *cond, struct lock *lock)`
  If any threads are waiting on `cond` (protected by monitor lock `lock`), then this function wakes up one

of them. If no threads are waiting, returns without performing any action. `lock` must be held before calling this function.

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

E.5 Optimization Barriers

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

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

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

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

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

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

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

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

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

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

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

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

```
intr_set_level (old_level);
```

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

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

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

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

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

F Advice

You should read through and understand as much of the Pintos source code that you mean to modify before starting work on project. In a sense, this is why we have you write a design document; it should be obvious that you have a good understanding, at the very least at a high level, of files such as `userprog/process.c`. We see groups in office hours who are really struggling due to a conceptual misunderstanding that has informed the way they designed their implementations and thus has caused bugs when trying to actually implement them in code.

You should learn to use the advanced features of GDB. Often times, debugging your code usually takes longer than writing it. However, a good understanding of the code you are modifying can help you pinpoint where the error might be; hence, again, we strongly recommend you to read through and understand at least the files you will be modifying in this project (with the caveat that it is a large codebase, so don't overwhelm yourself).

These projects are designed to be difficult and even push you to your limits as a systems programmer, so plan to be busy and have fun!

F.1 Group Work

In the past, many groups divided each assignment into pieces. Then, each group member worked on his or her piece until just before the deadline, at which time the group reconvened to combine their code and submit. This is a bad idea. We do not recommend this approach. Groups that do this often find that two changes conflict with each other, requiring lots of last-minute debugging. Some groups who have done this have turned in code that did not even compile or boot, much less pass any tests.

Instead, we recommend integrating your team's changes early and often, using git. This is less likely to produce surprises, because everyone can see everyone else's code as it is written, instead of just when it is finished. These systems also make it possible to review changes and, when a change introduces a bug, drop back to working versions of code.

We also encourage you to program in pairs, or even as a group. Having multiple sets of eyes looking at the same code can help avoid subtle bugs that would've otherwise been very difficult to debug.

F.1.1 Meetings

We encourage each group to have regular meetings (e.g. twice a week) to make sure everyone is on the same page. In-person meetings are generally much more productive, since people tend to be more attentive.

If you're meeting through a call (e.g. Zoom), we recommend you enable live transcription[4]. Moreover, you should take detailed notes during each meeting on a central document to reference back to.

F.2 Development

F.2.1 Compiler Warnings

Compiler warnings are your friend! When compiling your code, we have configured GCC to emit a variety of helpful warnings when it detects suspicious or problematic conditions in your code (e.g. using the value of an uninitialized variable, comparing two values of different types, etc). When you run `make` to compile your code, by default it echoes each command it is executing, which creates a huge amount of output that you usually don't care about, obscuring compiler warnings. To hide this output and show only compiler warnings (in addition to anything else printed to standard error by the commands `make` is running), you can run `make -s`. The `-s` flag tells `make` to be silent instead of echoing every command. **Do NOT pass the `-s` flag when running `make check`** or you won't see your test results! Only use the `-s` flag when compiling your code.

[4] https://support.zoom.us/hc/en-us/articles/207279736-Managing-closed-captioning-and-live-transcription

If your code is buggy, the first thing you should do is check to see if the compiler is emitting any warnings. While sometimes warnings might be emitted for code that is perfectly fine, in general it's best to remedy your code to fix warnings whenever you see them.

F.2.2 Faster Compilation

Depending on the machine you're using, compiling the Pintos code may take a while to complete. You can speed this up by using `make`'s `-j` flag to compile several files in parallel. For maximum effectiveness, the value provided for `-j` (which effectively specifies how many things to compile in parallel) should be equal to the number of (logical) CPUs on your machine which can be found by running the `nproc` command in your shell. You can combine all of this into one command by running `make -j $(nproc)` instead of running just `make` whenever you want to compile your code.

Please be warned however that **you should only pass the `-j` flag to `make` when compiling your code**. You should *not* use it when running your tests with `make check`. While it is actually safe to do so for Project User Programs, this is not the case for other projects, so it's best not to get into the habit of it.

F.2.3 Repeated Commands

You'll often find yourself having to type in the same/similar long commands (e.g. `PINTOS_DEBUG`, `loadusersymbols`). Instead of retyping these every time or copying and pasting, you can use reverse-i-search using `Ctrl-R`. This will alow you to quickly search through your command history. If there are multiple matches, you can cycle through them by pressing `Ctrl-R` repeatedly.

Within GDB, you can specifically shorten your commands which GDB will automatically match as long as there are no ambiguities. For instance, you can type `deb` instead of `debugpintos` and `n` instead of `next`. Moreover, pressing enter without typing any command will repeat the command, which is useful for stepping through the code.

F.2.4 Hail Mary

Rarely you may find yourself in a bizarre situation where the behavior of your kernel isn't changing even though you're certain you've changed your code in a way that should produce an obvious effect. If this occurs, you can destroy all compiled objects and caches, and restore your current terminal and shell parameters to sane values by running the following.

```
hash -r
stty sane
cd ~/code/group/pintos/src
make clean
```

Once you've done the above, recompile your code and try whatever it was you were doing again.