

CS 162 Project 1: Threads

Design Document Due:	Friday, February 19, 2016
Code Due:	Friday, March 4, 2016
Final Report Due:	Monday, March 7, 2016

Contents

1	Your task	3
1.1	Task 1: Efficient Alarm Clock	3
1.2	Task 2: Priority Scheduler	3
1.3	Task 3: Multi-level Feedback Queue Scheduler (MLFQS)	3
2	Deliverables	4
2.1	Design Document (Due 2/19) and Design Review	4
2.1.1	Design Document Guidelines	4
2.1.2	Design Document Additional Questions	6
2.1.3	Design Review	6
2.1.4	Grading	6
2.2	Code (Due 3/4)	7
2.3	Final Report (Due 3/7) and Code Quality	7
3	Reference	8
3.1	Pintos	8
3.1.1	Getting Started	8
3.1.2	Source Tree	8
3.1.3	Building Pintos	9
3.1.4	Running Pintos	10
3.1.5	Debugging Pintos	10
3.1.6	Debugging Pintos Tests	11
3.1.7	Debugging Page Faults	12
3.1.8	Debugging Kernel Panics	12
3.1.9	Adding Source Files	13
3.1.10	Why Pintos?	13
3.2	Threads	13
3.2.1	Understanding Threads	13
3.2.2	The Thread Struct	14
3.2.3	Thread Functions	16
3.2.4	Thread Switching	17
3.3	Synchronization	18
3.3.1	Disabling Interrupts	18
3.3.2	Semaphores	19
3.3.3	Locks	20
3.3.4	Monitors	21

3.3.5	Optimization Barriers	21
3.4	Memory Allocation	23
3.4.1	Page Allocator	23
3.4.2	Block Allocator	24
3.5	Linked Lists	25
3.6	Efficient Alarm Clock	26
3.7	Priority Scheduler	26
3.8	Advanced Scheduler	27
3.8.1	Introduction	27
3.8.2	Fixed-point Real Numbers	27
3.8.3	Niceness	28
3.8.4	Calculating Priority	28
3.8.5	Calculating Recent CPU	28
3.8.6	Calculating Load Average	29
3.8.7	Summary	30
3.8.8	Additional Details	30
3.9	Advice	30

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.

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. Your task is to re-implement `timer_sleep()` so that it executes efficiently without any “busy waiting”.

1.2 Task 2: Priority Scheduler

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

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. The purpose of priority donation is to solve the priority inversion problem. For example, a scheduler that did not donate priorities would choose medium-priority threads instead of B. However, we want B to run first, so we can unblock A. Your implementation of priority donation must handle 1) donations from multiple sources, 2) undoing donations when a lock is released, and 3) nested/recursive donation.

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

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

1.3 Task 3: Multi-level Feedback Queue Scheduler (MLFQS)

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

2 Deliverables

Your project grade will be made up of 3 components:

- 20% Design Document and Design Review
- 70% Code
- 10% Final Report and Code Quality

2.1 Design Document (Due 2/19) 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

Write your design document inside the `doc/project1.md` file, which has already been placed in your group's GitHub repository. You must use [GitHub Flavored Markdown](#)¹ to format your design document. You can preview your design document on GitHub's web interface by going to the following address: (replace group0 with your group number)

<https://github.com/Berkeley-CS162/group0/blob/master/doc/project1.md>

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 parts. 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

¹<https://help.github.com/articles/basic-writing-and-formatting-syntax/>

- Releasing a Lock
 - Computing the effective priority
 - Priority scheduling for semaphores and locks
 - Priority scheduling for condition variables
 - Changing thread's priority
- Use backticks around variable names and function names. Use bold, italics, and other Markdown styles to improve the readability of your design document.
 - 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. Consider a fully-functional correct implementation of this project, except for a single bug, which exists in the `sema_up()` function. According to the project requirements, semaphores (and other synchronization variables) must prefer higher-priority threads over lower-priority threads. However, my 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. (This question uses the MLFQS scheduler.) Suppose threads A, B, and C have nice values 0, 1, and 2. Each has a `recent_cpu` value of 0. Fill in the table below showing the scheduling decision and the `recent_cpu` and `priority` values for each thread after each given number of timer ticks. We can use `R(A)` and `P(A)` to denote the `recent_cpu` and `priority` values of thread A, for brevity.

timer ticks	R(A)	R(B)	R(C)	P(A)	P(B)	P(C)	thread to run
0							
4							
8							
12							
16							
20							
24							
28							
32							
36							

Helpful tip: A GitHub-flavored Markdown version of this table is available at <https://gist.github.com/rogerhub/82395ealf3ed64db6779>.

3. Did any ambiguities in the scheduler specification make values in the table (in the previous question) uncertain? If so, what rule did you use to resolve them?

2.1.3 Design Review

You will schedule a 20-25 minute design review with your project TA. During the design review, your TA will ask you questions about your design for the project. You should be prepared to defend your design and answer any clarifying questions your TA may have about your design document. The design review is also a good opportunity to get to know your TA for those participation points.

2.1.4 Grading

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

2.2 Code (Due 3/4)

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.

2.3 Final Report (Due 3/7) and Code Quality

After you complete the code for your project, you will submit a final report. Write your final report inside the `reports/project1.md` file, which has already been placed in your group's GitHub repository. Please include the following in your final report:

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

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

- Does your code exhibit any major memory safety problems (especially regarding C strings), memory leaks, poor error handling, or race conditions?
- Did you use consistent code style? Your code should blend in with the existing Pintos code. Check your use of indentation, your spacing, and your naming conventions.
- 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?
- Did you re-implement linked list algorithms instead of using the provided list manipulation
- 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

Pintos is an educational operating system for the x86 architecture. It supports kernel threads, loading and running user programs, and a file system, but it implements all of these in a very simple way. In the Pintos projects, you and your project team will strengthen its support in all three of these areas.

Pintos could, theoretically, run on a regular IBM-compatible PC. Unfortunately, it is impractical to supply every CS 162 student a dedicated PC for use with Pintos. Therefore, we will run Pintos projects in a system simulator, that is, a program that simulates an x86 CPU and its peripheral devices accurately enough that unmodified operating systems and software can run under it. Simulators also give you the ability to inspect and debug an operating system while it runs. In class we will use the Bochs and QEMU simulators.

3.1.1 Getting Started

Log in to the Vagrant Virtual Machine that you set up in hw0. You should already have a copy of the Pintos skeleton code in `~/code/group` on your VM. Since this is your first group project, you will need to link this local git repository to your group's GitHub repository. Go to the [“Group” section of the autograder²](#) and click the Octocat icon to go to your group's GitHub repository. Copy the **SSH Clone URL** (it should look like `git@github.com:Berkeley-CS162/groupX.git`) and use it in the commands below:

```
$ cd ~/code/group
$ git remote add group YOUR_GITHUB_CLONE_URL
```

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 1 tests:

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

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

3.1.2 Source Tree

`threads/`

The base Pintos kernel. Most of the modifications you will make for Project 1 will be in this directory.

`userprog/`

The user program loader. You will modify this for Project 2.

²<https://cs162.eecs.berkeley.edu/autograder/dashboard/group/>

vm/

We will not use this directory.

filesystem/

The Pintos file system. You will use this file system in Project 2 and modify it in Project 3.

devices/

Source code for I/O device interfacing: keyboard, timer, disk, etc. You will modify the timer implementation in Project 1.

lib/

An implementation of a subset of the C standard library. The code in this directory is compiled into both the Pintos kernel and user programs that run inside it. You can include header files from this directory using the `#include <...>` notation. You should not have to modify this code.

lib/kernel/

Library functions that are only included in the Pintos kernel (not the user programs). It contains implementations of some data types that you should use in your kernel code: bitmaps, doubly linked lists, and hash tables. In the kernel, headers in this directory can be included using the `#include <...>` notation.

lib/user/

Library functions that are included only in Pintos user programs (not the kernel). In user programs, headers in this directory can be included using the `#include <...>` notation.

tests/

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

examples/

Example user programs that you can use in Project 2.

misc/, utils/

These files help you run Pintos. You should not need to interact with them directly.

Makefile.build

Describes how to build the kernel. Modify this file if you would like to add source code. For more information, see Adding Source Files.

3.1.3 Building Pintos

This section describes the interesting files inside the **threads/build** directory.

threads/build/Makefile

A copy of **Makefile.build**. Don't change this file, because your changes will be overwritten if you `make clean` and re-compile. Make your changes to **Makefile.build** instead. For more information, see Adding Source Files.

threads/build/kernel.o

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

threads/build/kernel.bin

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

threads/build/loader.bin

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

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

3.1.4 Running Pintos

We've supplied a program for running Pintos in a simulator, called `pintos`, which should already be in your `PATH`. (If not, add `$HOME/.bin` to your `PATH`.)

The Pintos kernel takes a list of arguments, which tell the kernel what actions to perform. These actions are specified in the file `threads/init.c` on line 309 and look something like this.

```
static const struct action actions[] =
{
    {"run", 2, run_task},
#ifdef FILESYS
    {"ls", 1, fsutil_ls},
    {"cat", 2, fsutil_cat},
    {"rm", 2, fsutil_rm},
    {"extract", 1, fsutil_extract},
    {"append", 2, fsutil_append},
#endif
    {NULL, 0, NULL},
};
```

The preprocessor macro `FILESYS` was not defined when we built Pintos earlier (it will be enabled in Project 2 and Project 3). So, we can only take one action: `run`. This `run` action will run a test specified by the next command-line argument. The number next to each action's name tells Pintos how many arguments there are (including the name of the action itself). For example, the `run` action's arguments are "`run <test_name>`". The tests that you can run are the file names in `tests/threads` (without the file extensions).

Let's try it out! First `cd` into the `threads` directory. Then run "`pintos run alarm-multiple`", which passes the arguments "`run alarm-multiple`" to the Pintos kernel. In these arguments, `run` instructs the kernel to run a test named `alarm-multiple`.

3.1.5 Debugging Pintos

The `pintos` program offers several options for configuring the simulator or the virtual hardware. If you specify any options, they must precede the commands passed to the Pintos kernel and be separated from them by `--`, so that the whole command looks like "`pintos [option...] -- [argument...]`". Invoke `pintos` without any arguments to see a list of available options.

One of the most important options is `--gdb` which will allow you to use `gdb` to debug the code you've written. For example, to run the debugger for the `alarm-multiple` test we would perform the following steps:

- `cd` into `~/code/group/pintos/src/threads`
- Run `pintos` with the debugger option `"pintos --gdb -- run alarm-multiple"`. At this point, Pintos should say that it is "Waiting for gdb connection on port 1234".
- Open a new terminal and SSH into the VM
- `cd` into `~/code/group/pintos/src/threads/build`
- Open `cgdb` by running `"pintos-gdb kernel.o"`. The `pintos-gdb` script loads `cgdb` with many useful GDB macros that will help you debug Pintos.
- In GDB, attach to Pintos by running `"target remote localhost:1234"`. On your VM, you should be able to use `"debugpintos"` or `"deb"` as a shortcut to performing this step.
- Set a breakpoint at an interesting point in the program. Try `"break thread_init"`.
- Use `"continue"` or `"c"` to start Pintos.

When you are working with the Pintos test suite, you should not start Pintos by constructing the command-line arguments manually. Many of the tests in the test suite require specific arguments to be passed to the simulator or to Pintos itself. You should read the next section for information about how to debug Pintos tests.

3.1.6 Debugging Pintos Tests

To debug Pintos test cases, you should first run the `"make check"` command, then **copy the command-line arguments for the test that you are debugging**, and then add `--gdb` to it. Then, run `pintos-gdb` like you did in the previous section.

Try it out yourself. SSH into your VM and `cd` to `~/code/group/pintos/src/threads`. Then, run:

```
$ make clean
$ make
$ make check
```

Pay attention to the output of `"make check"`. You should see lines like these in the output:

```
pintos -v -k -T 60 --bochs -- -q run alarm-multiple < /dev/null 2> tests/threads...
perl -I../.. ../tests/threads/alarm-multiple.ck tests/threads/alarm-multiple te...
pass tests/threads/alarm-multiple
```

Here is an explanation:

- The first line runs the `pintos` script to start the Pintos kernel in a simulator, like we did before. The command uses the `-v -k -T 60 --bochs` options for the simulator, and the `-q run alarm-multiple` arguments for the Pintos kernel. Then, we use the `<`, `2>`, and `>` symbols to redirect standard input, standard error, and standard output to files.
- The second line uses the Perl programming language to run `../tests/threads/alarm-multiple.ck` to verify the output of Pintos kernel.
- Using the Perl script from line 2, the build system can tell if this test passed or failed. If the test passed, we will see "pass", followed by the test name. Otherwise, we will see "fail", followed by the test name, and then more details about the test failure.

In order to debug this test, you should copy the command on the first line. Remove the input/output redirection (everything after the “< /dev/null”), because we want to see the output on the terminal when we’re debugging. Finally, add `--gdb` to the simulator options. (The `--gdb` must be before the double dashes, `--`, because everything after the double dashes is passed to the kernel, not the simulator.)

Your final command should look like:

```
$ pintos --gdb -v -k -T 60 --bochs -- -q run alarm-multiple
```

Run this command. Then, open a new terminal and `cd` to `~/code/group/pintos/src/threads/build` and run “`pintos-gdb kernel.o`” and type “`debugpintos`” like you did in the previous section.

- You do not need to quit GDB after each run of the Pintos kernel. Just start `pintos` again and type `debugpintos` into GDB to attach to a new instance of Pintos. (If you re-compile your code, you must quit GDB and start it again, so it can load the new symbols from `kernel.o`.)
- Take some time to learn all the GDB shortcuts and how to use the CGDB interface. You may also be interested in looking at the Pintos GDB macros found in `~/bin/gdb-macros`.

3.1.7 Debugging Page Faults

If you encounter a page fault during a test, you should use the method in the previous section to debug Pintos with GDB. If you use `pintos-gdb` instead of plain `gdb` or `cgdb`, you should get a backtrace of the page fault when it occurs:

```
pintos-debug: a page fault occurred in kernel mode
#0 test_alarm_negative () at ../../tests/threads/alarm-negative.c:14
#1 0xc000ef4c in ?? ()
#2 0xc0020165 in start () at ../../threads/start.S:180
```

If you want to inspect the original environment where the page fault occurred, you can use this trick:

```
(gdb) debugpintos
(gdb) continue
```

Now, wait until the kernel encounters the page fault. Then run these commands:

```
(gdb) set $eip = ((void**) $esp)[1]
(gdb) up
(gdb) down
```

You should now be able to inspect the local variables and the stack trace when the page fault occurred.

3.1.8 Debugging Kernel Panics

The Pintos source code contains a lot of “`ASSERT (condition)`” statements. When the condition of an assert statement evaluates to false, the kernel will panic and print out some debugging information. Usually, you will get a line of output that looks like this:

```
Call stack: 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319
0xc010325a 0x804812c 0x8048a96 0x8048ac8.
```

This is a list of instruction addresses, each of which corresponds to a frame on the kernel stack when the panic occurred. You can decode this information into a helpful stack trace by using the `backtrace` utility that is included in your VM.

```
$ cd ~/code/group/pintos/src/threads/build/
$ backtrace kernel.o 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319 ...
```

If you run your tests using “`make check`”, the testing framework will run `backtrace` automatically when it detects a kernel panic.

To debug a kernel panic with GDB, you can usually just set a breakpoint at the inner-most line of code inside the stack trace. However, if your kernel panic occurs inside a function that is called many times, you may need to type `continue` a bunch of times before you reach the point in the test where the kernel panic occurs.

One trick you can use to improve this technique is to transform the code itself. For example, if you have an assert statement that looks like:

```
ASSERT (is_thread (next));
```

You can transform it into this:

```
if (!is_thread(next)) {
    barrier(); // Set a breakpoint HERE!
}
ASSERT (is_thread (next));
```

Then, set a breakpoint at the line containing `barrier()`. You can use any line of code instead of `barrier()`, but you must ensure that the compiler cannot reorder or eliminate that line of code. For example, if you created a dummy variable “`int hello = 1;`” instead of using `barrier()`, the compiler could decide that line of code wasn’t needed and omit instructions for it! If you get a compile error while using `barrier()`, make sure you’ve included the `synch.h` header file.

You can also use GDB’s conditional breakpoints, but if the assertion makes use of C macros, GDB might not understand what you mean.

3.1.9 Adding Source Files

This project will not require you to add any new source code files. In the event you want to add your own `.c` source code, open `Makefile.build` in your Pintos root directory and add the file to either the `threads_SRC` or `devices_SRC` variable depending on where the files are located.

If you want to add your own tests, place the test files in `tests/threads/`. Then, edit `tests/threads/Make.tests` to incorporate your tests into the build system.

Make sure to re-run `make` from the `threads` directory after adding your files. If your new file doesn’t get compiled, run `make clean` and try again. Note that adding new `.h` files will not require any changes to makefiles.

3.1.10 Why Pintos?

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

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.)

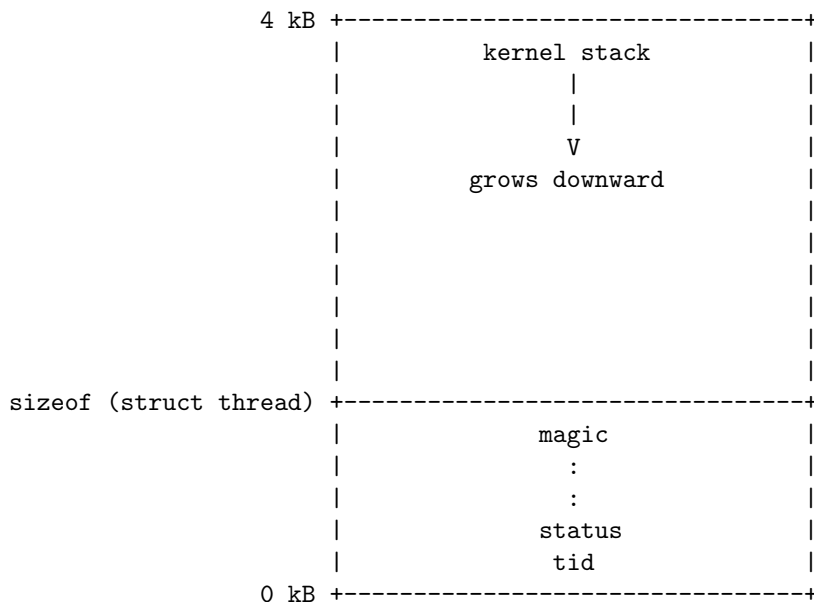
The mechanics of a context switch are in `threads/switch.S`, which is x86 assembly code. It saves the state of the currently running thread and restores the state of the next thread onto the CPU.

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: uint32_t *pagedir**
(Only used in Project 2 and later.) The page table for the process, if this is a user process.
- **Member of struct thread: unsigned magic**
Always set to `THREAD_MAGIC`, which is just an arbitrary number defined in `threads/thread.c`, and used to detect stack overflow. `thread_current()` checks that the `magic` member of the running thread’s `struct thread` is set to `THREAD_MAGIC`. Stack overflow tends to change this value, triggering the assertion. For greatest benefit, as you add members to `struct thread`, leave `magic` at the end.

3.2.3 Thread Functions

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

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

Before `thread_init()` runs, `thread_current()` will fail because the running thread’s `magic` value is incorrect. Lots of functions call `thread_current()` directly or indirectly, including `lock_acquire()` for locking a lock, so `thread_init()` is called early in Pintos initialization.
- **Function: void thread_start (void)**
Called by `main()` to start the scheduler. Creates the idle thread, that is, the thread that is scheduled when no other thread is ready. Then enables interrupts, which as a side effect enables the scheduler because the scheduler runs on return from the timer interrupt, using `intr_yield_on_return()`.
- **Function: void thread_tick (void)**
Called by the timer interrupt at each timer tick. It keeps track of thread statistics and triggers the scheduler when a time slice expires.
- **Function: void thread_print_stats (void)**
Called during Pintos shutdown to print thread statistics.
- **Function: tid_t thread_create (const char *name, int priority, thread_func *func, void *aux)**
Creates and starts a new thread named `name` with the given `priority`, returning the new thread’s `tid`. The thread executes `func`, passing `aux` as the function’s single argument.

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

– **Type: void thread_func (void *aux)**
This is the type of the function passed to `thread_create()`, whose `aux` argument is passed along as the function’s argument.
- **Function: void thread_block (void)**
Transitions the running thread from the running state to the blocked state. The thread will not run again until `thread_unblock()` is called on it, so you’d better have some way arranged for

that to happen. Because `thread_block()` is so low-level, you should prefer to use one of the synchronization primitives instead.

- **Function:** `void thread_unblock (struct thread *thread)`
Transitions `thread`, which must be in the blocked state, to the ready state, allowing it to resume running. This is called when the event that the thread is waiting for occurs, e.g. when the lock that the thread is waiting on becomes available.
- **Function:** `struct thread *thread_current (void)`
Returns the running thread.
- **Function:** `tid_t thread_tid (void)`
Returns the running thread's thread id. Equivalent to `thread_current ()->tid`.
- **Function:** `const char *thread_name (void)`
Returns the running thread's name. Equivalent to `thread_current ()->name`.
- **Function:** `void thread_exit (void) NO_RETURN`
Causes the current thread to exit. Never returns, hence `NO_RETURN`.
- **Function:** `void thread_yield (void)`
Yields the CPU to the scheduler, which picks a new thread to run. The new thread might be the current thread, so you can't depend on this function to keep this thread from running for any particular length of time.
- **Function:** `void thread_foreach (thread_action_func *action, void *aux)`
Iterates over all threads `t` and invokes `action(t, aux)` on each. `action` must refer to a function that matches the signature given by `thread_action_func()`:
 - **Type:** `void thread_action_func (struct thread *thread, void *aux)`
Performs some action on a thread, given `aux`.
- **Function:** `int thread_get_priority (void)`
Function: `void thread_set_priority (int new_priority)`
Stub to set and get thread priority.
- **Function:** `int thread_get_nice (void)`
Function: `void thread_set_nice (int new_nice)`
Function: `int thread_get_recent_cpu (void)`
Function: `int thread_get_load_avg (void)`
Stubs for the advanced MLFQS scheduler.

3.2.4 Thread Switching

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

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

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

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

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

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

3.3 Synchronization

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

3.3.1 Disabling Interrupts

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

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

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

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

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

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

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

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

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

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

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

3.3.2 Semaphores

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

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

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

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

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

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

- **Type:** `struct semaphore`
Represents a semaphore.

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

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

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

3.3.3 Locks

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

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

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

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

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

3.3.4 Monitors

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

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

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

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

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

Sending a signal and waking up from a wait are not an atomic operation. Thus, typically `cond_wait`’s caller must recheck the condition after the wait completes and, if necessary, wait again.
- **Function:** `void cond_signal (struct condition *cond, struct lock *lock)`
If any threads are waiting on `cond` (protected by monitor lock `lock`), then this function wakes up one of them. If no threads are waiting, returns without performing any action. `lock` must be held before calling this function.
- **Function:** `void cond_broadcast (struct condition *cond, struct lock *lock)`
Wakes up all threads, if any, waiting on `cond` (protected by monitor lock `lock`). `lock` must be held before calling this function.

3.3.5 Optimization Barriers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4 Memory Allocation

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

3.4.1 Page Allocator

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

The page allocator divides the memory it allocates into two pools, called the kernel and user pools. By default, each pool gets half of system memory above 1 MiB, but the division can be changed with the `-ul` kernel command line option. An allocation request draws from one pool or the other. If one pool becomes empty, the other may still have free pages. The user pool should be used for allocating memory for user processes and the kernel pool for all other allocations. This distinction will only become important starting with Project 2. Until then, all allocations should be made from the kernel pool.

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

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

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

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

Page allocator types and functions are described below.

- **Function:** `void * palloc_get_page (enum palloc_flags flags)`
Function: `void * palloc_get_multiple (enum palloc_flags flags, size_t page_cnt)`
 Obtains and returns one page, or `page_cnt` contiguous pages, respectively. Returns a null pointer if the pages cannot be allocated.

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

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

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

3.4.2 Block Allocator

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

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

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

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

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

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

The block allocator may not be called from interrupt context.

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

- **Function:** `void * malloc (size_t size)`
 Obtains and returns a new block, from the kernel pool, at least `size` bytes long. Returns a null pointer if `size` is zero or if memory is not available.
- **Function:** `void * calloc (size_t a, size_t b)`
 Obtains and returns a new block, from the kernel pool, at least `a * b` bytes long. The block's contents will be cleared to zeros. Returns a null pointer if `a` or `b` is zero or if insufficient memory is available.
- **Function:** `void * realloc (void *block, size_t new_size)`
 Attempts to resize `block` to `new_size` bytes, possibly moving it in the process. If successful, returns the new block, in which case the old block must no longer be accessed. On failure, returns a null pointer, and the old block remains valid.
 A call with `block` null is equivalent to `malloc`. A call with `new_size` zero is equivalent to `free`.
- **Function:** `void free (void *block)`
 Frees `block`, which must have been previously returned by `malloc`, `calloc`, or `realloc` (and not yet freed).

3.5 Linked Lists

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.6 Efficient Alarm Clock

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

1. If `timer_sleep()` is called with a zero or negative argument, then you should just return immediately.
2. When you run Pintos, the clock does **not** run in realtime by default. So, if a thread goes to sleep for 5 “seconds” (e.g. ticks = $5 \times \text{TIMER_FREQ}$), it will actually be much shorter than 5 seconds in terms of wall clock time. You can use the `--realtime` flag for Pintos to override this.
3. Separate functions `timer_msleep()`, `timer_usleep()`, and `timer_nsleep()` do exist for sleeping a specific number of milliseconds, microseconds, or nanoseconds, respectively, but these will call `timer_sleep()` automatically when necessary. You do not need to modify them.
4. The code that runs in interrupt handlers (i.e. `timer_interrupt()`) should be as fast as possible. It’s usually wise to do some pre-computation outside of the interrupt handler, in order to make the interrupt handler as fast as possible. Additionally, you may not acquire locks while executing `timer_interrupt()`.
5. Pay close attention to the Pintos linked-list implementation. Each linked list requires a dedicated `list_elem` member inside its elements. Every element of a linked list should be the same type. If you create new linked lists, make sure that they are initialized. Finally, make sure that there are no race conditions for any of your linked lists (the list manipulation functions are **NOT** thread-safe).

3.7 Priority Scheduler

Here are some more details about the Priority Scheduler task.

1. A thread’s initial priority is an argument of `thread_create()`. You should use `PRI_DEFAULT` (31), unless there is a reason to use a different value.
2. Your implementation must handle nested donation: Consider a high-priority thread H, a medium-priority thread M, and a low-priority thread L. If H must wait on M and M must wait on L, then we should donate H’s priority to L.
3. A thread can only donate to 1 thread at a time, because once it calls `lock_acquire()`, the donor thread is blocked.

4. If there are multiple waiters on a lock when you call `lock_release()`, then all of those priority donations must apply to the thread that receives the lock next.
5. You do not need to handle priority values outside of the allowed range, `PRI_MIN` (0) to `PRI_MAX` (63).
6. You only need to implement priority donation for locks. Do not implement them for other synchronization variables (it doesn't make any sense to do it for semaphores or monitors anyway). However, you need to implement priority scheduling for locks, semaphores, and condition variables. Priority scheduling is when you unblock the highest priority thread when a resource is released or a monitor is signaled.
7. Don't forget to implement `thread_get_priority()`, which is the function that returns the current thread's priority. This function should NOT take donations into account. You should return the base priority of the thread.
8. A thread cannot change another thread's priority, except via donations. The `thread_set_priority()` function only acts on the current thread.
9. If a thread no longer has the highest effective priority (e.g. because it released a lock or it called `thread_set_priority()` with a lower value), it must immediately yield the CPU. If a lock is released, but the current thread still has the highest effective priority, it should not yield the CPU.

3.8 Advanced Scheduler

Here are some more details about the Advanced Scheduler task.

3.8.1 Introduction

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

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

Your advanced scheduler is a **multilevel feedback queue** scheduler. This type of scheduler maintains several queues of ready-to-run threads, where each queue holds threads with a different priority. The scheduler always chooses a thread from the highest-priority non-empty queue. If the highest-priority queue contains multiple threads, then they run in "round robin" order.

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

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

3.8.2 Fixed-point Real Numbers

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

`fixed-point.h` to represent any value that needs to be a real number. If you use integers, your values will not be correct.

3.8.3 Niceness

Each thread has an integer `nice` value that determines how “nice” the thread should be to other threads. A `nice` of zero does not affect thread priority. A positive `nice` (to the maximum of 20) decreases the priority of a thread and causes it to give up some CPU time it would otherwise receive. On the other hand, a negative `nice` (to the minimum of -20) tends to take away CPU time from other threads.

The initial thread starts with a `nice` value of zero. Other threads start with a `nice` value inherited from their parent thread. You must implement the functions described below, which are for use by the test framework. We have provided skeleton definitions for them in “`threads/thread.c`”.

- **Function:** `int thread_get_nice (void)`
Returns the current thread’s `nice` value.
- **Function:** `void thread_set_nice (int new_nice)`
Sets the current thread’s `nice` value to `new_nice` and recalculates the thread’s priority based on the new value. If the running thread no longer has the highest priority, it should yield the CPU.

3.8.4 Calculating Priority

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

$$\text{priority} = \text{PRI_MAX} - (\text{recent_cpu}/4) - (\text{nice} \times 2)$$

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

This formula is designed so that threads that have recently been scheduled on the CPU will have a lower priority the next time the scheduler picks a thread to run. This is key to preventing starvation: a thread that has not received any CPU time recently will have a `recent_cpu` of 0, which barring a very high `nice` value, should ensure that it receives CPU time soon.

3.8.5 Calculating Recent CPU

We wish `recent_cpu` to measure how much CPU time each process has received “recently.” One approach would use an array of n elements to track the CPU time received in each of the last n seconds. However, this approach requires $O(n)$ space per thread and $O(n)$ time per calculation of a new weighted average.

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

$$\begin{aligned} x(0) &= f(0) \\ x(t) &= a \times x(t-1) + f(t) \\ a &= k/(k+1) \end{aligned}$$

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

$$\begin{aligned}
x(1) &= f(1) \\
x(2) &= a \times f(1) + f(2) \\
x(3) &= a^2 \times f(1) + a \times f(2) + f(3) \\
x(4) &= a^3 \times f(1) + a^2 \times f(2) + a \times f(3) + f(4)
\end{aligned}$$

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

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

$$\text{recent_cpu} = (2 \times \text{load_avg}) / (2 \times \text{load_avg} + 1) \times \text{recent_cpu} + \text{nice}$$

In this formula, `load_avg` is a moving average of the number of threads ready to run (see the next section). If `load_avg` is 1, indicating that a single thread, on average, is competing for the CPU, then the current value of `recent_cpu` decays to a weight of 0.1 in $\ln(0.1)/\ln(\frac{2}{3}) = \text{approx. 6 seconds}$; if `load_avg` is 2, then decay to a weight of 0.1 takes $\ln(0.1)/\ln(\frac{3}{4}) = 0.8 \text{ seconds}$. The effect is that `recent_cpu` estimates the amount of CPU time the thread has received “recently,” with the rate of decay inversely proportional to the number of threads competing for the CPU.

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

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

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

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

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

3.8.6 Calculating Load Average

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

$$\text{load_avg} = (59/60) \times \text{load_avg} + (1/60) \times \text{ready_threads}$$

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

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

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

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

3.8.7 Summary

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

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

$$\text{priority} = \text{PRI_MAX} - (\text{recent_cpu}/4) - (\text{nice} \times 2)$$

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

$$\text{recent_cpu} = (2 \times \text{load_avg}) / (2 \times \text{load_avg} + 1) \times \text{recent_cpu} + \text{nice}$$

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

$$\text{load_avg} = (59/60) \times \text{load_avg} + (1/60) \times \text{ready_threads}$$

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

3.8.8 Additional Details

1. When the advanced scheduler is enabled, you should **NOT** do priority donation.
2. When the advanced scheduler is enabled, threads no longer directly control their own priorities. The priority argument to `thread_create()` should be ignored, as well as any calls to `thread_set_priority()`, and `thread_get_priority()` should return the thread’s current priority as set by the scheduler.
3. Because many of these formulas involve fractions, you should use **fixed-point real arithmetic** for your calculations. Use the `fixed_point_t` type and the library functions inside `pintos/src/threads/fixed-point.h` to do your advanced scheduler calculations.

3.9 Advice

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

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

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

You should learn to use the advanced features of GDB. For this project, debugging your code usually takes longer than writing it.

Do not commit/push binary files or unneeded log files.

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

CS 162 Project 2: User Programs

Initial Design Document Due:

Monday, March 28, 2016

Code Due:

Monday, April 11, 2016

Final Report Due:

Monday, April 11, 2016

Contents

1	Your task	3
1.1	Task 1: Argument Passing	3
1.2	Task 2: Process Control Syscalls	3
1.3	Task 3: File Operation Syscalls	3
2	Deliverables	4
2.1	Design Document (Due 3/28) and Design Review	4
2.1.1	Design Document Guidelines	4
2.1.2	Design Document Additional Questions	5
2.1.3	Design Review	5
2.1.4	Grading	5
2.2	Code (Due 4/11)	6
2.3	Student Testing (Due 4/11)	6
2.4	Final Report (Due 4/11) and Code Quality	7
3	Reference	8
3.1	Pintos	8
3.1.1	Getting Started	8
3.1.2	Source Files	8
3.1.3	Using the File System	9
3.1.4	Formatting and Using the Filesystem	9
3.1.5	How User Programs Work	10
3.1.6	Virtual Memory Layout	10
3.1.7	Accessing User Memory	11
3.1.8	80x86 Calling Convention	12
3.1.9	Program Startup Details	13
3.1.10	Adding new tests to Pintos	14
3.2	System Calls	15
3.2.1	System Call Overview	15
3.2.2	Process System Calls	15
3.2.3	File System Calls	16
3.3	FAQ	17

3.3.1	Argument Passing FAQ	19
3.3.2	System Calls FAQ	19
3.4	Debugging Tips	20
3.4.1	printf	20
3.4.2	ASSERT	20
3.4.3	Function and parameter attributes	20
3.4.4	Backtraces	21
3.4.5	GDB	23
3.4.6	Triple Faults	27
3.4.7	General Tips	28

1 Your task

In this project, you will extend Pintos's support for user programs. The skeleton code for Pintos is already able to load and start user programs, but the programs cannot read command-line arguments or make system calls.

1.1 Task 1: Argument Passing

The “`process_execute(char *file_name)`” function is used to create new user-level processes in Pintos. Currently, it does not support command-line arguments. You must implement argument passing, so that calling “`process_execute("ls -ahl")`” will provide the 2 arguments, [`"ls"`, `"-ahl"`], to the user program using `argc` and `argv`.

All of our Pintos test programs start by printing out their own name (e.g. `argv[0]`). Since argument passing has not yet been implemented, all of these programs will crash when they access `argv[0]`. Until you implement argument passing, none of the user programs will work.

1.2 Task 2: Process Control Syscalls

Pintos currently only supports one syscall: `exit`. You will add support for the following new syscalls: `halt`, `exec`, `wait`, and `practice`. Each of these syscalls has a corresponding function inside the user-level library, `lib/user/syscall.c`, which prepares the syscall arguments and handles the transfer to kernel mode. The kernel's syscall handlers are located in `userprog/syscall.c`.

The `halt` syscall will shutdown the system. The `exec` syscall will start a new program with `process_execute()`. (There is no `fork` syscall in Pintos. The Pintos `exec` syscall is similar to calling Linux's `fork` syscall and then Linux's `execve` syscall in the child process immediately afterward.) The `wait` syscall will wait for a specific child process to exit. The `practice` syscall just adds 1 to its first argument, and returns the result (to give you practice writing a syscall handler).

To implement syscalls, you first need a way to safely read and write memory that's in user process's virtual address space. The syscall arguments are located on the user process's stack, right above the user process stack pointer. If the stack pointer is invalid when the user program makes a syscall, the kernel cannot crash while trying to dereference an invalid or null pointer. Additionally, some syscall arguments are pointers to buffers inside the user process's address space. Those buffer pointer could be invalid as well.

You will need to gracefully handle cases where a syscall cannot be completed, because of invalid memory access. These kinds of memory errors include null pointers, invalid pointers (which point to unmapped memory locations), or pointers to the kernel's virtual address space. Beware: It may be the case that a 4-byte memory region (like an 32-bit integer) consists of 2 bytes of valid memory and 2 bytes of invalid memory, if the memory lies on a page boundary. You should handle these cases by terminating the user process. We recommend testing this part of your code, before implementing any other system call functionality. See 3.1.7 Accessing User Memory for more information.

1.3 Task 3: File Operation Syscalls

In addition to the process control syscalls, you will also need to implement these file operation syscalls: `create`, `remove`, `open`, `filesize`, `read`, `write`, `seek`, `tell`, and `close`. Pintos already contains a basic filesystem. Your implementation of these syscalls will simply call the appropriate functions in the file system library. You will not need to implement any of these file operations yourself.

The Pintos filesystem is not thread-safe. You must make sure that your file operation syscalls do not call multiple filesystem functions concurrently. In Project 3, you will add more sophisticated synchronization to the Pintos filesystem, but for this project, you are permitted to use a global lock on filesystem operations, to ensure thread safety. We recommend that you avoid modifying the `filesystem/` directory in this project.

While a user process is running, you must ensure that nobody can modify its executable on disk. The “rox” tests ensure that you deny writes to current-running program files. The functions `file_deny_write()` and `file_allow_write()` can assist with this feature.

Note: Your final code for Project 2 will be used as a starting point for Project 3. The tests for Project 3 depend on some of the same syscalls that you are implementing for this project. You should keep this in mind while designing your implementation for this project.

2 Deliverables

Your project grade will be made up of 4 components:

- 20% Design Document and Design Review
- 55% Code
- **15% Student Testing** – This is a new component of your project grade (see explanation below).
- 10% Final Report and Code Quality

2.1 Design Document (Due 3/28) 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

Write your design document inside the `doc/project2.md` file, which has already been placed in your group’s GitHub repository. You must use GitHub Flavored Markdown¹ to format your design document. You can preview your design document on GitHub’s web interface by going to the following address: (replace `group0` with your group number)

<https://github.com/Berkeley-CS162/group0/blob/master/doc/project2.md>

For each of the first 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 parts. 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**. This section should be similar in style and format to the design document you submitted for Project 1. We expect you to read through the Pintos source code when preparing your design document, and your design document should refer to the Pintos source when necessary to clarify your implementation.

¹<https://help.github.com/articles/basic-writing-and-formatting-syntax/>

3. **Synchronization** – This section should list all resources that are shared across threads. For each case, enumerate how the resources are accessed (e.g., from inside of the scheduler, in an interrupt context, etc), and describe the strategy you plan to use to ensure that these resources are shared and modified safely. For each resource, demonstrate that your design ensures correct behavior and avoids deadlock. 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. Please discuss the time/memory costs of your synchronization approach, and whether your strategy will significantly limit the parallelism of the kernel. When discussing the parallelism 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.
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. Take a look at the Project 2 test suite in `pintos/src/tests/userprog`. Some of the test cases will intentionally provide invalid pointers as syscall arguments, in order to test whether your implementation safely handles the reading and writing of user process memory. Please identify a test case that uses an **invalid** stack pointer (`$esp`) when making a syscall. Provide the name of the test and explain how the test works. (Your explanation should be very specific: use line numbers and the actual names of variables when explaining the test case.)
2. Please identify a test case that uses a **valid** stack pointer when making a syscall, but the stack pointer is too close to a page boundary, so some of the syscall arguments are located in invalid memory. (Your implementation should kill the user process in this case.) Provide the name of the test and explain how the test works. (Your explanation should be very specific: use line numbers and the actual names of variables when explaining the test case.)
3. Identify **one** part of the project requirements which is **not fully tested by the existing test suite**. Explain what kind of test needs to be added to the test suite, in order to provide coverage for that part of the project. (There are multiple good answers for this question.)

2.1.3 Design Review

You will schedule a 20-25 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. Your score will reflect how convincing your design is, based on your explanation in your design document and your answers during the design review. You **must** attend a design review in order to get these points. We will try to accommodate any time conflicts, but you should let your TA know as soon as possible.

2.2 Code (Due 4/11)

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.

2.3 Student Testing (Due 4/11)

Pintos already contains a test suite for Project 2, but not all of the parts of this project have complete test coverage. **Your task is to submit 2 new test cases, which exercise functionality that is not covered by existing tests.** We will not tell you what features to write tests for. 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 15 characters.

Once you finish writing your test cases, make sure that they get executed when you run “`make check`” in the `pintos/src/userprog/` directory. You will need to prepare a Student Testing Report, which will help us grade your test cases. Place your Student Testing Report inside `reports/project2.md`, alongside your final report.

Make sure your Student Testing Report contains the following:

- For each of the 2 test cases you write:
 - Provide a description of the feature your test case is supposed to test.
 - Provide an overview of how the mechanics of your test case work, as well as a qualitative description of the expected output.
 - Provide the output of your own Pintos kernel when you run the test case. Please copy the full raw output file from `userprog/build/tests/userprog/your-test-1.output` as well as the raw results from `userprog/build/tests/userprog/your-test-1.result`.
 - Identify two non-trivial potential kernel bugs, and explain how they would have affected your output of this test case. You should express these in this form: “If your 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. “If your kernel had a syntax error, then this test case would not run.” does not count).
- Tell us about your experience writing tests for Pintos. What can be improved about the Pintos testing system? (There’s a lot of room for improvement.) What did you learn from writing test cases?

We will grade your test cases based on effort. If all of the above components are present in your Student Testing Report and your test cases are satisfactory, you will get full credit on this part of the project.

2.4 Final Report (Due 4/11) and Code Quality

After you complete the code for your project, you will submit a final report. Write your final report inside the `reports/project2.md` file, which has already been placed in your group's GitHub repository. 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?
- Your Student Testing Report (see the previous section for more details)

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

- Does your code exhibit any major memory safety problems (especially regarding C strings), memory leaks, poor error handling, or race conditions?
- Did you use consistent code style? Your code should blend in with the existing Pintos code. Check your use of indentation, your spacing, and your naming conventions.
- 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?
- Did you re-implement linked list algorithms instead of using the provided list manipulation
- 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

User programs are written under the illusion that they have the entire machine, which means that the operating system must manage/protect machine resources correctly to maintain this illusion for multiple processes. In Pintos, more than one process can run at a time, but each process is single-threaded (multithreaded processes are not supported).

3.1.1 Getting Started

Log in to the Vagrant Virtual Machine that you set up in hw0. You should already have your Pintos code from Project 1 in `~/code/group` on your VM. You may start Project 2 using your Project 1 code. But you may also start over from the skeleton code if you wish.

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

```
$ cd ~/code/group/
$ git checkout 448edfde949b4ce35f30cc49af350e7a44401815 -- pintos/
$ git commit -m "Revert changes to pintos/ from Project 1"
$ git push group master
```

Make sure **your design document and final report for Project 1 is still accessible** from the GitHub website. We will be looking for those documents, so if you do not have them, we might mistakenly think you didn't turn them in. Also, please **do not force push** and please do not delete your commits from Project 1. You should know that orphan commits are still accessible on GitHub, and we have a history of the commit hashes you've pushed to the autograder, but we will not enjoy digging up that information if we need it.

We recommend that you use Git to tag your final Project 1 code, for your own benefit.

Once you have made some progress on your project, you can push your code to the autograder by pushing to "group master". 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/userprog
$ 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.

3.1.2 Source Files

For this project, you will need to modify some of the same files that you changed in Project 1. You may also need to modify these additional files:

process.c Loads ELF binaries and starts processes.

pagedir.c Manages the page tables. You probably won't need to modify this code, but you may want to call some of these functions.

userprog/syscall.c This is a skeleton system call handler. Currently, it only supports the `exit` syscall.

lib/user/syscall.c Provides library functions for user programs to invoke system calls from a C program. Each function uses inline assembly code to prepare the syscall arguments and invoke the system call. We don't expect you to understand this assembly code, but we do expect you to understand the calling conventions used for syscalls (also in Reference).

lib/syscall-nr.h This file defines the syscall numbers for each syscall.

exception.c Handle exceptions. Currently all exceptions simply print a message and terminate the process. Some, but not all, solutions to Project 2 involve modifying `page_fault()` in this file.

3.1.3 Using the File System

You will need to use the Pintos filesystem for this project, in order to load user programs from disk and implement file operation syscalls. You will not need to modify the filesystem in this project. The provided filesystem already contains all the functionality needed to support the required syscalls. (We recommend that you do not change the filesystem for this project.) However, you will need to read some of the filesystem code, especially `filesys.h` and `file.h`, to understand how to use the file system. You should beware of these limitations of the Pintos filesystem:

- No internal synchronization. Concurrent accesses will interfere with one another. You should use synchronization to ensure that only one process at a time is executing file system code.
- File size is fixed at creation time. The root directory is represented as a file, so the number of files that may be created is also limited.
- File data is allocated as a single extent. In other words, data in a single file must occupy a contiguous range of sectors on disk. External fragmentation can therefore become a serious problem as a file system is used over time.
- No subdirectories.
- File names are limited to 14 characters.
- A system crash mid-operation may corrupt the disk in a way that cannot be repaired automatically. There is no file system repair tool anyway.
- When a file is removed (deleted), its blocks are not deallocated until all processes have closed all file descriptors pointing to it. Therefore, a deleted file may still be accessible by processes that have it open.

3.1.4 Formatting and Using the Filesystem

During the development process, you may need to be able to create a simulated disk with a file system partition. The `pintos-mkdisk` program provides this functionality. From the `userprog/build` directory, execute `pintos-mkdisk filesys.dsk`

`--filesys-size=2`. This command creates a simulated disk named `filesys.dsk` that contains a 2 MB Pintos file system partition. Then format the file system partition by passing `-f -q` on the kernel's command line: `pintos -f -q`. The `-f` option causes the file system to be formatted, and `-q` causes Pintos to exit as soon as the format is done.

You'll need a way to copy files in and out of the simulated file system. The Pintos `-p` ("put") and `-g` ("get") options do this. To copy file into the Pintos file system, use the command `pintos -p file -- -q`. (The `--` is needed because `-p` is for the Pintos script, not for the simulated kernel.) To copy it to the Pintos file system under the name `newname`, add `-a newname`: `pintos -p file -a newname -- -q`. The commands for copying files out of a VM are similar, but substitute `-g` for `-p`.

Here's a summary of how to create a disk with a file system partition, format the file system, copy the echo program into the new disk, and then run echo, passing argument x. (Argument passing won't work until you implemented it.) It assumes that you've already built the examples in examples and that the current directory is userprog/build:

```
pintos-mkdisk filesys.dsk --fileysys-size=2
pintos -f -q
pintos -p ../../examples/echo -a echo -- -q
pintos -q run 'echo x'
```

The three final steps can actually be combined into a single command:

```
pintos-mkdisk filesys.dsk --fileysys-size=2
pintos -p ../../examples/echo -a echo -- -f -q run 'echo x'
```

If you don't want to keep the file system disk around for later use or inspection, you can even combine all four steps into a single command. The `--fileysys-size=n` option creates a temporary file system partition approximately n megabytes in size just for the duration of the Pintos run. The Pintos automatic test suite makes extensive use of this syntax:

```
pintos --fileysys-size=2 -p ../../examples/echo -a echo -- -f -q run 'echo x'
```

You can delete a file from the Pintos file system using the `rm` file kernel action, e.g. `pintos -q rm file`. Also, `ls` lists the files in the file system and `cat file` prints a file's contents to the display.

3.1.5 How User Programs Work

Pintos can run normal C programs, as long as they fit into memory and use only the system calls you implement. Notably, `malloc()` cannot be implemented because none of the system calls required for this project allow for memory allocation. Pintos also can't run programs that use floating point operations, since the kernel doesn't save and restore the processor's floating-point unit when switching threads.

The `src/examples` directory contains a few sample user programs. The Makefile in this directory compiles the provided examples, and you can edit it to compile your own programs as well. Pintos can load *ELF* executables with the loader provided for you in `userprog/process.c`.

Until you copy a test program to the simulated file system (see Using the File System), Pintos will be unable to do useful work. You should create a clean reference file system disk and copy that over whenever you trash your `fileysys.dsk` beyond a useful state, which may happen occasionally while debugging.

3.1.6 Virtual Memory Layout

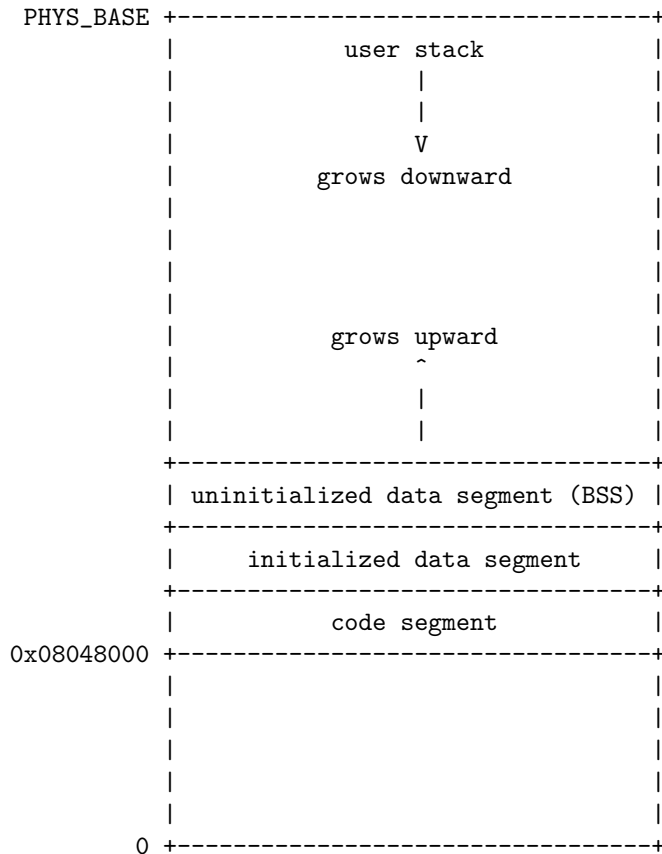
Virtual memory in Pintos is divided into two regions: user virtual memory and kernel virtual memory. User virtual memory ranges from virtual address 0 up to `PHYS_BASE`, which is defined in `threads/vaddr.h` and defaults to `0xC0000000` (3 GB). Kernel virtual memory occupies the rest of the virtual address space, from `PHYS_BASE` up to 4 GB.

User virtual memory is per-process. When the kernel switches from one process to another, it also switches user virtual address spaces by changing the processor's page directory base register (see `pagedir_activate()` in `userprog/pagedir.c`). `struct thread` contains a pointer to a process's page table.

Kernel virtual memory is global. It is always mapped the same way, regardless of what user process or kernel thread is running. In Pintos, kernel virtual memory is mapped one-to-one to physical memory, starting at `PHYS_BASE`. That is, virtual address `PHYS_BASE` accesses physical address 0, virtual address `PHYS_BASE + 0x1234` accesses physical address `0x1234`, and so on up to the size of the machine's physical memory.

A user program can only access its own user virtual memory. An attempt to access kernel virtual memory causes a page fault, handled by `page_fault()` in `userprog/exception.c`, and the process will be terminated. Kernel threads can access both kernel virtual memory and, if a user process is running, the user virtual memory of the running process. However, even in the kernel, an attempt to access memory at an unmapped user virtual address will cause a page fault.

Typical Memory Layout Conceptually, each process is free to lay out its own user virtual memory however it chooses. In practice, user virtual memory is laid out like this:



3.1.7 Accessing User Memory

As part of a system call, the kernel must often access memory through pointers provided by a user program. The kernel must be very careful about doing so, because the user can pass a null pointer, a pointer to unmapped virtual memory, or a pointer to kernel virtual address space (above `PHYS_BASE`). All of these types of invalid pointers must be rejected without harm to the kernel or other running processes, by terminating the offending process and freeing its resources.

There are at least two reasonable ways to do this correctly:

- verify the validity of a user-provided pointer, then dereference it. If you choose this route, you'll want to look at the functions in `userprog/pagedir.c` and in `threads/vaddr.h`. This is the simplest way to handle user memory access.
- check only that a user pointer points below `PHYS_BASE`, then dereference it. An invalid user pointer will cause a “page fault” that you can handle by modifying the code for `page_fault()` in

userprog/exception.c. This technique is normally faster because it takes advantage of the processor's MMU, so it tends to be used in real kernels (including Linux).

In either case, you need to make sure not to “leak” resources. For example, suppose that your system call has acquired a lock or allocated memory with `malloc()`. If you encounter an invalid user pointer afterward, you must still be sure to release the lock or free the page of memory. If you choose to verify user pointers before dereferencing them, this should be straightforward. It's more difficult to handle if an invalid pointer causes a page fault, because there's no way to return an error code from a memory access. Therefore, for those who want to try the latter technique, we'll provide a little bit of helpful code:

```
/* Reads a byte at user virtual address UADDR.
   UADDR must be below PHYS_BASE.
   Returns the byte value if successful, -1 if a segfault
   occurred. */
static int
get_user (const uint8_t *uaddr)
{
    int result;
    asm ("movl $1f, %0; movzbl %1, %0; 1:"
        : "=a" (result) : "m" (*uaddr));
    return result;
}

/* Writes BYTE to user address UDST.
   UDST must be below PHYS_BASE.
   Returns true if successful, false if a segfault occurred. */
static bool
put_user (uint8_t *udst, uint8_t byte)
{
    int error_code;
    asm ("movl $1f, %0; movb %b2, %1; 1:"
        : "=a" (error_code), "=m" (*udst) : "q" (byte));
    return error_code != -1;
}
```

Each of these functions assumes that the user address has already been verified to be below `PHYS_BASE`. They also assume that you've modified `page_fault()` so that a page fault in the kernel merely sets `%eax` to `0xffffffff` and copies its former value into `eip`.

3.1.8 80x86 Calling Convention

This section summarizes important points of the convention used for normal function calls on 32-bit 80x86 implementations of Unix. Some details are omitted for brevity.

The calling convention works like this:

1. The caller pushes each of the function's arguments on the stack one by one, normally using the `PUSH` assembly language instruction. Arguments are pushed in right-to-left order.

The stack grows downward: each push decrements the stack pointer, then stores into the location it now points to, like the C expression `*--sp = value`.

2. The caller pushes the address of its next instruction (the *return address*) on the stack and jumps to the first instruction of the callee. A single 80x86 instruction, `CALL`, does both.
3. The callee executes. When it takes control, the stack pointer points to the return address, the first argument is just above it, the second argument is just above the first argument, and so on.
4. If the callee has a return value, it stores it into register `EAX`.
5. The callee returns by popping the return address from the stack and jumping to the location it specifies, using the 80x86 `RET` instruction.
6. The caller pops the arguments off the stack.

Consider a function `f()` that takes three `int` arguments. This diagram shows a sample stack frame as seen by the callee at the beginning of step 3 above, supposing that `f()` is invoked as `f(1, 2, 3)`. The initial stack address is arbitrary:

	+-----+
0xbffffe7c	3
0xbffffe78	2
0xbffffe74	1
stack pointer --> 0xbffffe70	return address
	+-----+

3.1.9 Program Startup Details

The Pintos C library for user programs designates `_start()`, in `lib/user/entry.c`, as the entry point for user programs. This function is a wrapper around `main()` that calls `exit()` if `main()` returns:

```
void
_start (int argc, char *argv[])
{
    exit (main (argc, argv));
}
```

The kernel must put the arguments for the initial function on the stack before it allows the user program to begin executing. The arguments are passed in the same way as the normal calling convention (see section 80x86 Calling Convention).

Consider how to handle arguments for the following example command: `/bin/ls -l foo bar`. First, break the command into words: `/bin/ls`, `-l`, `foo`, `bar`. Place the words at the top of the stack. Order doesn't matter, because they will be referenced through pointers.

Then, push the address of each string plus a null pointer sentinel, on the stack, in right-to-left order. These are the elements of `argv`. The null pointer sentinel ensures that `argv[argc]` is a null pointer, as required by the C standard. The order ensures that `argv[0]` is at the lowest virtual address. Word-aligned accesses are faster than unaligned accesses, so for best performance round the stack pointer down to a multiple of 4 before the first push.

Then, push `argv` (the address of `argv[0]`) and `argc`, in that order. Finally, push a fake "return address": although the entry function will never return, its stack frame must have the same structure as any other.

The table below shows the state of the stack and the relevant registers right before the beginning of the user program, assuming `PHYS_BASE` is `0xc0000000`:

Address	Name	Data	Type
0xbfffffc	argv[3][...]	bar\0	char[4]
0xbfffff8	argv[2][...]	foo\0	char[4]
0xbfffff5	argv[1][...]	-l\0	char[3]
0xbffffed	argv[0][...]	/bin/ls\0	char[8]
0xbffffec	word-align	0	uint8_t
0xbffffe8	argv[4]	0	char *
0xbffffe4	argv[3]	0xbfffffc	char *
0xbffffe0	argv[2]	0xbfffff8	char *
0xbffffdc	argv[1]	0xbfffff5	char *
0xbffffd8	argv[0]	0xbffffed	char *
0xbffffd4	argv	0xbffffd8	char **
0xbffffd0	argc	4	int
0xbffffcc	return address	0	void (*)()

In this example, the stack pointer would be initialized to 0xbffffcc.

As shown above, your code should start the stack at the very top of the user virtual address space, in the page just below virtual address `PHYS_BASE` (defined in `threads/vaddr.h`).

You may find the non-standard `hex_dump()` function, declared in `<stdio.h>`, useful for debugging your argument passing code. Here's what it would show in the above example:

```

bfffffc0                                00 00 00 00 |          ....|
bfffffd0  04 00 00 00 d8 ff ff bf-ed ff ff bf f5 ff ff bf |.....|
bfffffe0  f8 ff ff bf fc ff ff bf-00 00 00 00 2f 62 69 |...../bi|
bffffff0  6e 2f 6c 73 00 2d 6c 00-66 6f 6f 00 62 61 72 00 |n/ls.-l.foo.bar.|

```

3.1.10 Adding new tests to Pintos

Pintos also comes with its own testing framework that allows you to design and run your own tests. For this project, you will also be required to extend the current suite of tests with a few tests of your own. All of the filesystem and userprog tests are “user program” tests, which means that they are only allowed to interact with the kernel via system calls.

Some things to keep in mind while writing your test cases:

- User programs have access to a limited subset of the C standard library. You can find the user library in `lib/`.
- User programs cannot directly access variables in the kernel.
- User programs do not have access to `malloc`, since `brk` and `sbrk` are not implemented. User programs also have a limited stack size. If you need a large buffer, make it a static global variable.
- Pintos starts with 4MB of memory and the file system block device is 2MB by default. Don't use data structures or files that exceed these sizes.
- Your test should use `msg()` instead of `printf()` (they have the same function signature).

You can add new test cases to the `userprog` suite by modifying these files:

tests/userprog/Make.tests Entry point for the `userprog` test suite. You need to add the name of your test to the `tests/userprog_TESTS` variable, in order for the test suite to find it. Additionally, you will need to define a variable named `tests/userprog/my-test-1_SRC` which contains all the files that need to be compiled into your test (see the other test definitions for examples). You can add other source files and resources to your tests, if you wish.

tests/userprog/my-test-1.c This is the test code for your test. Your test should define a function called `test_main`, which contains a user-level program. This is the main body of your test case, which should make syscalls and print output. Use the `msg()` function instead of `printf`.

tests/userprog/my-test-1.ck Every test needs a `.ck` file, which is a Perl script that checks the output of the test program. If you are not familiar with Perl, don't worry! You can probably get through this part with some educated guessing. Your check script should use the subroutines that are defined in `tests/tests.pm`. At the end, call `pass` to print out the "PASS" message, which tells Pintos test driver that your test passed.

3.2 System Calls

3.2.1 System Call Overview

The first project already dealt with one way that the operating system can regain control from a user program: interrupts from timers and I/O devices. These are "external" interrupts, because they are caused by entities outside the CPU. The operating system also deals with software exceptions, which are events that occur in program code. These can be errors such as a page fault or division by zero. Exceptions are also the means by which a user program can request services ("system calls") from the operating system.

In the 80x86 architecture, the `int` instruction is the most commonly used means for invoking system calls. This instruction is handled in the same way as other software exceptions. In Pintos, user programs invoke `int $0x30` to make a system call. The system call number and any additional arguments are expected to be pushed on the stack in the normal fashion before invoking the interrupt (see section 80x86 Calling Convention).

Thus, when the system call handler `syscall_handler()` gets control, the system call number is in the 32-bit word at the caller's stack pointer, the first argument is in the 32-bit word at the next higher address, and so on. The caller's stack pointer is accessible to `syscall_handler()` as the `esp` member of the `struct intr_frame` passed to it. (`struct intr_frame` is on the kernel stack.)

The 80x86 convention for function return values is to place them in the EAX register. System calls that return a value can do so by modifying the `eax` member of `struct intr_frame`.

You should try to avoid writing large amounts of repetitive code for implementing system calls. Each system call argument, whether an integer or a pointer, takes up 4 bytes on the stack. You should be able to take advantage of this to avoid writing much near-identical code for retrieving each system call's arguments from the stack.

3.2.2 Process System Calls

For task 2, you will need to implement the following system calls:

System Call: `int practice (int i)` A "fake" system call that doesn't exist in any modern operating system. You will implement this to get familiar with the system call interface. This system call increments the passed in integer argument by 1 and returns it to the user.

System Call: `void halt (void)` Terminates Pintos by calling `shutdown_power_off()` (declared in `devices/shutdown.h`). This should be seldom used, because you lose some information about possible deadlock situations, etc.

System Call: `void exit (int status)` Terminates the current user program, returning status to the kernel. If the process's parent waits for it (see below), this is the status that will be returned. Conventionally, a status of 0 indicates success and nonzero values indicate errors. Every user program that finishes in the normal way calls `exit` - even a program that returns from `main()` calls `exit` indirectly (see `start()` in `lib/user/entry.c`).

System Call: `pid_t exec (const char *cmd_line)` Runs the executable whose name is given in `cmd_line`, passing any given arguments, and returns the new process's program id (`pid`). Must return `pid -1`, which otherwise should not be a valid `pid`, if the program cannot load or run for any reason. Thus, the parent process cannot return from the `exec` until it knows whether the child process successfully loaded its executable. You must use appropriate synchronization to ensure this.

System Call: `int wait (pid_t pid)` Waits for a child process `pid` and retrieves the child's exit status.

If `pid` is still alive, waits until it terminates. Then, returns the status that `pid` passed to `exit`. If `pid` did not call `exit()`, but was terminated by the kernel (e.g. killed due to an exception), `wait(pid)` must return `-1`. It is perfectly legal for a parent process to wait for child processes that have already terminated by the time the parent calls `wait`, but the kernel must still allow the parent to retrieve its child's exit status, or learn that the child was terminated by the kernel.

`wait` must fail and return `-1` immediately if any of the following conditions is true:

- `pid` does not refer to a direct child of the calling process. `pid` is a direct child of the calling process if and only if the calling process received `pid` as a return value from a successful call to `exec`.

Note that children are not inherited: if A spawns child B and B spawns child process C, then A cannot wait for C, even if B is dead. A call to `wait(C)` by process A must fail. Similarly, orphaned processes are not assigned to a new parent if their parent process exits before they do.

- The process that calls `wait` has already called `wait` on `pid`. That is, a process may wait for any given child at most once.

Processes may spawn any number of children, wait for them in any order, and may even exit without having waited for some or all of their children. Your design should consider all the ways in which waits can occur. All of a process's resources, including its struct thread, must be freed whether its parent ever waits for it or not, and regardless of whether the child exits before or after its parent.

You must ensure that Pintos does not terminate until the initial process exits. The supplied Pintos code tries to do this by calling `process_wait()` (in `userprog/process.c`) from `main()` (in `threads/init.c`). We suggest that you implement `process_wait()` according to the comment at the top of the function and then implement the `wait` system call in terms of `process_wait()`.

Warning: Implementing this system call requires considerably more work than any of the rest.

3.2.3 File System Calls

For task 3, you will need to implement the following system calls:

System Call: `bool create (const char *file, unsigned initial_size)` Creates a new file called `file` initially `initial_size` bytes in size. Returns true if successful, false otherwise. Creating a new file does not open it: opening the new file is a separate operation which would require a `open` system call.

System Call: `bool remove (const char *file)` Deletes the file called `file`. Returns true if successful, false otherwise. A file may be removed regardless of whether it is open or closed, and removing an open file does not close it. See Removing an Open File, for details.

System Call: `int open (const char *file)` Opens the file called `file`. Returns a nonnegative integer handle called a “file descriptor” (`fd`), or `-1` if the file could not be opened.

File descriptors numbered 0 and 1 are reserved for the console: `fd 0 (STDIN_FILENO)` is standard input, `fd 1 (STDOUT_FILENO)` is standard output. The `open` system call will never return either of these file descriptors, which are valid as system call arguments only as explicitly described below.

Each process has an independent set of file descriptors. File descriptors are not inherited by child processes.

When a single file is opened more than once, whether by a single process or different processes, each `open` returns a new file descriptor. Different file descriptors for a single file are closed independently in separate calls to `close` and they do not share a file position.

System Call: `int filesize (int fd)` Returns the size, in bytes, of the file open as `fd`.

System Call: `int read (int fd, void *buffer, unsigned size)` Reads `size` bytes from the file open as `fd` into `buffer`. Returns the number of bytes actually read (0 at end of file), or `-1` if the file could not be read (due to a condition other than end of file). `Fd 0` reads from the keyboard using `input_getc()`.

System Call: `int write (int fd, const void *buffer, unsigned size)` Writes `size` bytes from `buffer` to the open file `fd`. Returns the number of bytes actually written, which may be less than `size` if some bytes could not be written.

Writing past end-of-file would normally extend the file, but file growth is not implemented by the basic file system. The expected behavior is to write as many bytes as possible up to end-of-file and return the actual number written, or 0 if no bytes could be written at all.

`Fd 1` writes to the console. Your code to write to the console should write all of `buffer` in one call to `putbuf()`, at least as long as `size` is not bigger than a few hundred bytes. (It is reasonable to break up larger buffers.) Otherwise, lines of text output by different processes may end up interleaved on the console, confusing both human readers and our grading scripts.

System Call: `void seek (int fd, unsigned position)` Changes the next byte to be read or written in open file `fd` to `position`, expressed in bytes from the beginning of the file. (Thus, a `position` of 0 is the file’s start.)

A seek past the current end of a file is not an error. A later read obtains 0 bytes, indicating end of file. A later write extends the file, filling any unwritten gap with zeros. (However, in Pintos files have a fixed length until project 4 is complete, so writes past end of file will return an error.) These semantics are implemented in the file system and do not require any special effort in system call implementation.

System Call: `unsigned tell (int fd)` Returns the position of the next byte to be read or written in open file `fd`, expressed in bytes from the beginning of the file.

System Call: `void close (int fd)` Closes file descriptor `fd`. Exiting or terminating a process implicitly closes all its open file descriptors, as if by calling this function for each one.

3.3 FAQ

How much code will I need to write? Here’s a summary of our reference solution, produced by the `diffstat` program. the final row gives total lines inserted and deleted; a changed line counts as both an insertion and a deletion.

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.

```

threads/thread.c      |   13
threads/thread.h      |   26 +
userprog/exception.c  |    8
userprog/process.c    |  247 ++++++++--
userprog/syscall.c    |  468 ++++++++
userprog/syscall.h    |    1
6 files changed, 725 insertions(+), 38 deletions(-)

```

The kernel always panics when I run `pintos -p file -- -q`. Did you format the file system (with `pintos -f`)?

Is your file name too long? The file system limits file names to 14 characters. A command like `pintos -p ../../examples/echo -- -q` will exceed the limit. Use `pintos -p ../../examples/echo -a echo -- -q` to put the file under the name `echo` instead.

Is the file system full?

Does the file system already contain 16 files? The base Pintos file system has a 16-file limit.

The file system may be so fragmented that there's not enough contiguous space for your file.

When I run `pintos -p ../file --`, file isn't copied. Files are written under the name you refer to them, by default, so in this case the file copied in would be named `../file`. You probably want to run `pintos -p ../file -a file --` instead.

You can list the files in your file system with `pintos -q ls`.

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 with system call! You'll have to implement system calls before you see anything else. Every reasonable program tries to make at least one system call (`exit()`) and most programs make more than that. Notably, `printf()` invokes the write system call. The default system call handler just prints system call! and terminates the program. Until then, you can use `hex_dump()` to convince yourself that argument passing is implemented correctly (see section Program Startup Details).

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 `disassemble` 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 system calls for I/O and memory allocation. (Not all functions do, of course, but usually the library is compiled as a unit.)

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

Can I run user programs under a debugger? Yes, with some limitations. See the section of this spec on GDB macros.

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` system calls.

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.

3.3.1 Argument Passing FAQ

Isn't the top of stack in kernel virtual memory? The top of stack is at `PHYS_BASE`, typically `0xc0000000`, which is also where kernel virtual memory starts. But before the processor pushes data on the stack, it decrements the stack pointer. Thus, the first (4-byte) value pushed on the stack will be at address `0xbfffffff`.

Is `PHYS_BASE` fixed? No. You should be able to support `PHYS_BASE` values that are any multiple of `0x10000000` from `0x80000000` to `0xf0000000`, simply via recompilation.

How do I handle multiple spaces in an argument list? Multiple spaces should be treated as one space. You do not need to support quotes or any special characters other than space.

Can I enforce a maximum size on the arguments list? You can set a reasonable limit on the size of the arguments.

3.3.2 System Calls FAQ

Can I just cast a struct `file *` to get a file descriptor?

Can I just cast a struct `thread *` to a `pid_t`? You will have to make these design decisions yourself. Most operating systems do distinguish between file descriptors (or pids) and the addresses of their kernel data structures. You might want to give some thought as to why they do so before committing yourself.

Can I set a maximum number of open files per process? It is better not to set an arbitrary limit. You may impose a limit of 128 open files per process, if necessary.

What happens when an open file is removed? You should implement the standard Unix semantics for files. That is, when a file is removed any process which has a file descriptor for that file may continue to use that descriptor. This means that they can read and write from the file. The file will not have a name, and no other processes will be able to open it, but it will continue to exist until all file descriptors referring to the file are closed or the machine shuts down.

How can I run user programs that need more than 4 kB stack space? You may modify the stack setup code to allocate more than one page of stack space for each process. This is not required in this project.

What should happen if an exec fails midway through loading? `exec` should return -1 if the child process fails to load for any reason. This includes the case where the load fails part of the way through the process (e.g. where it runs out of memory in the multi-oom test). Therefore, the parent process cannot return from the `exec` system call until it is established whether the load was successful or not. The child must communicate this information to its parent using appropriate synchronization, such as a semaphore, to ensure that the information is communicated without race conditions.

3.4 Debugging Tips

Many tools lie at your disposal for debugging Pintos. This section introduces you to a few of them.

3.4.1 `printf`

Don't underestimate the value of `printf`. The way `printf` is implemented in Pintos, you can call it from practically anywhere in the kernel, whether it's in a kernel thread or an interrupt handler, almost regardless of what locks are held.

`printf` is useful for more than just examining data. It can also help figure out when and where something goes wrong, even when the kernel crashes or panics without a useful error message. The strategy is to sprinkle calls to `printf` with different strings (e.g.: "<1>", "<2>", ...) throughout the pieces of code you suspect are failing. If you don't even see <1> printed, then something bad happened before that point, if you see <1> but not <2>, then something bad happened between those two points, and so on. Based on what you learn, you can then insert more `printf` calls in the new, smaller region of code you suspect. Eventually you can narrow the problem down to a single statement. See section Triple Faults, for a related technique.

3.4.2 `ASSERT`

Assertions are useful because they can catch problems early, before they'd otherwise be noticed. Ideally, each function should begin with a set of assertions that check its arguments for validity. (Initializers for functions' local variables are evaluated before assertions are checked, so be careful not to assume that an argument is valid in an initializer.) You can also sprinkle assertions throughout the body of functions in places where you suspect things are likely to go wrong. They are especially useful for checking loop invariants.

Pintos provides the `ASSERT` macro, defined in `<debug.h>`, for checking assertions.

`ASSERT (expression)` Tests the value of `expression`. If it evaluates to zero (false), the kernel panics. The panic message includes the expression that failed, its file and line number, and a backtrace, which should help you to find the problem. See Backtraces, for more information.

3.4.3 Function and parameter attributes

These macros defined in `<debug.h>` tell the compiler special attributes of a function or function parameter. Their expansions are GCC-specific.

`UNUSED` Appended to a function parameter to tell the compiler that the parameter might not be used within the function. It suppresses the warning that would otherwise appear.

`NO_RETURN` Appended to a function prototype to tell the compiler that the function never returns. It allows the compiler to fine-tune its warnings and its code generation.

`NO_INLINE` Appended to a function prototype to tell the compiler to never emit the function in-line. Occasionally useful to improve the quality of backtraces (see below).

PRINTF_FORMAT (format, first) Appended to a function prototype to tell the compiler that the function takes a `printf`-like format string as the argument numbered **format** (starting from 1) and that the corresponding value arguments start at the argument numbered **first**. This lets the compiler tell you if you pass the wrong argument types.

3.4.4 Backtraces

When the kernel panics, it prints a “backtrace,” that is, a summary of how your program got where it is, as a list of addresses inside the functions that were running at the time of the panic. You can also insert a call to `debug_backtrace`, prototyped in `<debug.h>`, to print a backtrace at any point in your code. `debug_backtrace_all`, also declared in `<debug.h>`, prints backtraces of all threads.

The addresses in a backtrace are listed as raw hexadecimal numbers, which are difficult to interpret. We provide a tool called `backtrace` to translate these into function names and source file line numbers. Give it the name of your `kernel.o` as the first argument and the hexadecimal numbers composing the backtrace (including the `0x` prefixes) as the remaining arguments. It outputs the function name and source file line numbers that correspond to each address.

If the translated form of a backtrace is garbled, or doesn’t make sense (e.g.: function A is listed above function B, but B doesn’t call A), then it’s a good sign that you’re corrupting a kernel thread’s stack, because the backtrace is extracted from the stack. Alternatively, it could be that the `kernel.o` you passed to `backtrace` is not the same kernel that produced the backtrace.

Sometimes backtraces can be confusing without any corruption. Compiler optimizations can cause surprising behavior. When a function has called another function as its final action (a tail call), the calling function may not appear in a backtrace at all. Similarly, when function A calls another function B that never returns, the compiler may optimize such that an unrelated function C appears in the backtrace instead of A. Function C is simply the function that happens to be in memory just after A.

Here’s an example. Suppose that Pintos printed out this following call stack:

```
Call stack: 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319
0xc010325a 0x804812c 0x8048a96 0x8048ac8.
```

You would then invoke the `backtrace` utility like shown below, cutting and pasting the backtrace information into the command line. This assumes that `kernel.o` is in the current directory. You would of course enter all of the following on a single shell command line, even though that would overflow our margins here:

```
backtrace kernel.o 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319 0xc010325a
0x804812c 0x8048a96 0x8048ac8
```

The backtrace output would then look something like this:

```
0xc0106eff: debug_panic (lib/debug.c:86)
0xc01102fb: file_seek (filesys/file.c:405)
0xc010dc22: seek (userprog/syscall.c:744)
0xc010cf67: syscall_handler (userprog/syscall.c:444)
0xc0102319: intr_handler (threads/interrupt.c:334)
0xc010325a: intr_entry (threads/intr-stubs.s:38)
0x804812c: (unknown)
0x8048a96: (unknown)
0x8048ac8: (unknown)
```

The first line in the backtrace refers to `debug_panic`, the function that implements kernel panics. Because backtraces commonly result from kernel panics, `debug_panic` will often be the first function shown in a backtrace.

The second line shows `file_seek` as the function that panicked, in this case as the result of an assertion failure. In the source code tree used for this example, line 405 of `fileSYS/file.c` is the assertion

```
assert (file_ofs >= 0);
```

(This line was also cited in the assertion failure message.) Thus, `file_seek` panicked because it passed a negative file offset argument.

The third line indicates that `seek` called `file_seek`, presumably without validating the offset argument. In this submission, `seek` implements the `seek` system call.

The fourth line shows that `syscall_handler`, the system call handler, invoked `seek`.

The fifth and sixth lines are the interrupt handler entry path.

The remaining lines are for addresses below `phys_base`. This means that they refer to addresses in the user program, not in the kernel. If you know what user program was running when the kernel panicked, you can re-run `backtrace` on the user program, like so: (typing the command on a single line, of course):

```
backtrace tests/filesys/extended/grow-too-big 0xc0106eff 0xc01102fb 0xc010dc22
0xc010cf67 0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
```

The results look like this:

```
0xc0106eff: (unknown)
0xc01102fb: (unknown)
0xc010dc22: (unknown)
0xc010cf67: (unknown)
0xc0102319: (unknown)
0xc010325a: (unknown)
0x804812c: test_main (...xtended/grow-too-big.c:20)
0x8048a96: main (tests/main.c:10)
0x8048ac8: _start (lib/user/entry.c:9)
```

You can even specify both the kernel and the user program names on the command line, like so:

```
backtrace kernel.o tests/filesys/extended/grow-too-big 0xc0106eff 0xc01102fb 0xc010dc22
0xc010cf67 0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
```

The result is a combined backtrace:

in kernel.o:

```
0xc0106eff: debug_panic (lib/debug.c:86)|
0xc01102fb: file_seek (fileSYS/file.c:405)|
0xc010dc22: seek (userprog/syscall.c:744)|
0xc010cf67: syscall_handler (userprog/syscall.c:444)|
0xc0102319: intr_handler (threads/interrupt.c:334)|
0xc010325a: intr_entry (threads/intr-stubs.s:38)|
```

in tests/filesys/extended/grow-too-big:

```
0x804812c: test_main (...xtended/grow-too-big.c:20)|
0x8048a96: main (tests/main.c:10)|
0x8048ac8: _start (lib/user/entry.c:9)|
```

Here's an extra tip for anyone who read this far: **backtrace** is smart enough to strip the **call stack:** header and "." trailer from the command line if you include them. This can save you a little bit of trouble in cutting and pasting. Thus, the following command prints the same output as the first one we used:

```
backtrace kernel.o call stack: 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319
0xc010325a 0x804812c 0x8048a96 0x8048ac8.
```

3.4.5 GDB

You can run Pintos under the supervision of the gdb debugger. First, start Pintos with the **--gdb** option, e.g.: **pintos --gdb -- run mytest**. Second, open a second terminal on the same machine and use **pintos-gdb** to invoke gdb on **kernel.o**

```
pintos-gdb kernel.o
```

and issue the following gdb command:

```
target remote localhost:1234
```

Now gdb is connected to the simulator over a local network connection. You can now issue any normal gdb commands. If you issue the **c** command, the simulated bios will take control, load Pintos, and then Pintos will run in the usual way. You can pause the process at any point with **ctrl+c**.

Using GDB

You can read the gdb manual by typing **info gdb** at a terminal command prompt. Here's a few commonly useful gdb commands:

c Continues execution until **ctrl+c** or the next breakpoint.

break function

break file:line

break *address Sets a breakpoint at **function**, at **line** within **file**, or **address**. (use a **0x** prefix to specify an address in hex.)

Use **break main** to make gdb stop when Pintos starts running.

p expression Evaluates the given **expression** and prints its value. If the expression contains a function call, that function will actually be executed.

l *address Lists a few lines of code around **address**. (use a **0x** prefix to specify an address in hex.)

bt Prints a stack backtrace similar to that output by the **backtrace** program described above.

p/a address Prints the name of the function or variable that occupies **address**. (use a **0x** prefix to specify an address in hex.)

disassemble function disassembles **function**.

We also provide a set of macros specialized for debugging Pintos, written by Godmar Back (gback@cs.vt.edu). You can type **help user-defined** for basic help with the macros. Here is an overview of their functionality, based on Godmar's documentation:

debugpintos Attach debugger to a waiting Pintos process on the same machine. Shorthand for `target remote localhost:1234`.

dumplist &list type element Prints the elements of `list`, which should be a `struct list` that contains elements of the given `type` (without the word `struct`) in which `element` is the `struct list_elem` member that links the elements.

Example: `dumplist &all_list thread allelem` prints all elements of `struct thread` that are linked in `struct list all_list` using the `struct list_elem allelem` which is part of `struct thread`.

btthread thread Shows the backtrace of `thread`, which is a pointer to the `struct thread` of the thread whose backtrace it should show. For the current thread, this is identical to the `bt` (backtrace) command. It also works for any thread suspended in `schedule`, provided you know where its kernel stack page is located.

btthreadlist list element shows the backtraces of all threads in `list`, the `struct list` in which the threads are kept. Specify `element` as the `struct list_elem` field used inside `struct_thread` to link the threads together.

Example: `btthreadlist all_list allelem` shows the backtraces of all threads contained in `struct list all_list`, linked together by `allelem`. This command is useful to determine where your threads are stuck when a deadlock occurs. Please see the example scenario below.

btthreadall short-hand for `btthreadlist all_list allelem`.

btpagefault Print a backtrace of the current thread after a page fault exception. Normally, when a page fault exception occurs, gdb will stop with a message that might say:

```
program received signal 0, signal 0.
0xc0102320 in intr0e_stub ()
```

In that case, the `bt` command might not give a useful backtrace. Use `btpagefault` instead.

You may also use `btpagefault` for page faults that occur in a user process. In this case, you may wish to also load the user program's symbol table using the `loadusersymbols` macro, as described above.

hook-stop GDB invokes this macro every time the simulation stops, which Bochs will do for every processor exception, among other reasons. If the simulation stops due to a page fault, `hook-stop` will print a message that says and explains further whether the page fault occurred in the kernel or in user code.

If the exception occurred from user code, `hook-stop` will say:

```
pintos-debug: a page fault exception occurred in user mode
pintos-debug: hit 'c' to continue, or 's' to step to intr_handler
```

In project 2, a page fault in a user process leads to the termination of the process. You should expect those page faults to occur in the robustness tests where we test that your kernel properly terminates processes that try to access invalid addresses. To debug those, set a break point in `page_fault` in `exception.c`, which you will need to modify accordingly.

If the page fault did not occur in user mode while executing a user process, then it occurred in kernel mode while executing kernel code. In this case, `hook-stop` will print this message:

```
pintos-debug: a page fault occurred in kernel mode
```

Followed by the output of the `btpagefault` command.

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

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

First, I start Pintos:

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

```
writing command line to /tmp/gdalqtb5uf.dsk...
```

```
Bochs -q
```

```
=====
```

```
Bochs x86 emulator 2.2.5
```

```
build from cvs snapshot on december 30, 2005
```

```
=====
```

```
000000000000i[      ] reading configuration from Bochssrc.txt
000000000000i[      ] enabled gdbstub
000000000000i[      ] installing nogui module as the Bochs gui
000000000000i[      ] 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.

loadusersymbols You can also use `gdb` to debug a user program running under Pintos. To do that, use the `loadusersymbols` macro to load the program’s symbol table:

```
loadusersymbol program
```

Where `program` is the name of the program’s executable (in the host file system, not in the Pintos file system). For example, you may issue:

```
(gdb) loadusersymbols tests/userprog/exec-multiple
add symbol table from file "tests/userprog/exec-multiple" at
.text_addr = 0x80480a0
```

After this, you should be able to debug the user program the same way you would the kernel, by placing breakpoints, inspecting data, etc. Your actions apply to every user program running in Pintos, not just to the one you want to debug, so be careful in interpreting the results: `GDB` does not know which process is currently active (because that is an abstraction the Pintos kernel creates). Also, a name that appears in both the kernel and the user program will actually refer to the kernel name. (The latter problem can be avoided by giving the user executable name on the `gdb` command line, instead of `kernel.o`, and then using `loadusersymbols` to load `kernel.o`.) `loadusersymbols` is implemented via `gdb`’s `add-symbol-file` command.

3.4.6 Triple Faults

When a CPU exception handler, such as a page fault handler, cannot be invoked because it is missing or defective, the CPU will try to invoke the “double fault” handler. If the double fault handler is itself missing or defective, that’s called a “triple fault.” a triple fault causes an immediate cpu reset.

Thus, if you get yourself into a situation where the machine reboots in a loop, that’s probably a “triple fault.” In a triple fault situation, you might not be able to use `printf` for debugging, because the reboots might be happening even before everything needed for `printf` is initialized.

There are at least two ways to debug triple faults. First, you can run Pintos in Bochs under GDB. If Bochs has been built properly for Pintos, a triple fault under GDB will cause it to print the message “triple fault: stopping for gdb” on the console and break into the debugger. (If Bochs is not running under GDB, a triple fault will still cause it to reboot.) You can then inspect where Pintos stopped, which is where the triple fault occurred.

Another option is what I call “debugging by infinite loop.” Pick a place in the Pintos code, insert the infinite loop `for (;;) ;` there, and recompile and run. There are two likely possibilities:

The machine hangs without rebooting. If this happens, you know that the infinite loop is running. That means that whatever caused the reboot must be after the place you inserted the infinite loop. Now move the infinite loop later in the code sequence.

The machine reboots in a loop. If this happens, you know that the machine didn’t make it to the infinite loop. Thus, whatever caused the reboot must be before the place you inserted the infinite loop. Now move the infinite loop earlier in the code sequence.

If you move around the infinite loop in a “binary search” fashion, you can use this technique to pin down the exact spot that everything goes wrong. It should only take a few minutes at most.

3.4.7 General Tips

The page allocator in `threads/palloc.c` and the block allocator in `threads/malloc.c` clear all the bytes in memory to `0xcc` at time of free. Thus, if you see an attempt to dereference a pointer like `0xcccccccc`, or some other reference to `0xcc`, there's a good chance you're trying to reuse a page that's already been freed. Also, byte `0xcc` is the cpu opcode for "invoke interrupt 3," so if you see an error like `interrupt 0x03 (#bp breakpoint exception)`, then Pintos tried to execute code in a freed page or block.

An assertion failure on the expression `sec_no < d->capacity` indicates that Pintos tried to access a file through an inode that has been closed and freed. Freeing an inode clears its starting sector number to `0xcccccccc`, which is not a valid sector number for disks smaller than about 1.6 TB.

CS 162 Project 3: File Systems

Design Document Due:	Friday, April 22, 2016
Code Due:	Thursday, May 4, 2016
Final Report Due:	Friday, May 5, 2016

Contents

1	Your task	2
1.1	Task 1: Buffer cache	2
1.2	Task 2: Extensible files	2
1.3	Task 3: Subdirectories	2
1.4	Synchronization requirement	2
2	Deliverables	3
2.1	Design Document (Due 4/22) and Design Review	3
2.1.1	Design Document Guidelines	3
2.1.2	Topics for your design document	4
2.1.3	Design Document Additional Questions	4
2.1.4	Design Review	5
2.1.5	Grading	5
2.2	Code (Due 5/4)	5
2.3	Student Testing (Due 5/5)	5
2.4	Final Report (Due 5/5) and Code Quality	6
3	Reference	7
3.1	Getting Started	7
3.2	Source Files	7
3.3	Testing File System Persistence	7
3.4	Requirements	8
3.4.1	Buffer Cache	8
3.4.2	Indexed and Extensible Files	8
3.4.3	Subdirectories	9
3.4.4	System Calls	10
3.5	Pintos user program tests	11
3.6	How to add tests to Pintos	11
3.7	Suggested Order of Implementation	12
3.8	FAQ	12

1 Your task

In this project, you will add 3 new features to the Pintos file system.

1.1 Task 1: Buffer cache

The functions `inode_read_at()` and `inode_write_at()` currently access the filesystem's underlying block device directly, each time you call them. Your task is to add a buffer cache for the filesystem, to improve the performance of reads and writes. Your buffer cache will cache individual disk blocks, so that (1) you can respond to reads with cached data and (2) you can coalesce multiple writes into a single disk operation. The buffer cache should have a maximum capacity of 64 disk blocks. You may choose a block replacement policy, but it must be at least as good as the clock algorithm. The buffer cache must be a **write-back cache**, not a write-through cache. You must make sure that ALL disk operations use your buffer cache, not just the two inode functions mentioned earlier.

1.2 Task 2: Extensible files

Pintos currently cannot extend the size of files, because the Pintos filesystem allocates each file as a single contiguous set of blocks. Your task is to modify the Pintos filesystem to support extending files. For this task, you must use an inode structure with direct, indirect, and doubly-indirect pointers, like the Unix file system does. The maximum file size you need to support is 8MiB (2^{23} bytes). You must also add support for a new syscall, `inumber(int fd)`, which returns the unique inode number of file associated with a particular file descriptor.

1.3 Task 3: Subdirectories

The current Pintos filesystem supports directories, but user programs have no way of using them (files can only be placed in the root directory right now). You must add the following syscalls to allow user programs to manipulate directories: `chdir`, `mkdir`, `readdir`, and `isdir`. You must also update the following syscalls so that they work with directories: `open`, `close`, `exec`, `remove`, and `inumber`. You must also add support for **relative paths** for any syscall with a file path argument. For example, if a process calls `chdir("my_files/")` and then `open("notes.txt")`, you should search for `notes.txt` relative to the current directory and open the file `my_files/notes.txt`. You also need to support absolute paths like `open("/my_files/notes.txt")`. You need to support the special `."` and `.."` names, when they appear in file path arguments, such as `open("../logs/foo.txt")`. Child processes should inherit the parent's current working directory. The first user process should have the root directory as its current working directory.

1.4 Synchronization requirement

Your project code should always be thread-safe, but for Project 3, we don't want you to just use a single global lock around the entire filesystem. As you plan your design for this project, **you must ensure that 2 operations acting on different disk sectors, different files, and different directories can run simultaneously**. You may serialize operations that occur on the same disk sector, the same file, or the same directory. For example, two syscalls like `remove("/my_files/notes.txt")` and `readdir(open("/my_files/"))` may exhibit mutual exclusion, because they involve the same directory. However, `read(open("/my_files/notes.txt"))` and `write(open("/my_files/test.c"))` should be allowed to run concurrently, since they operate on two different files and reading and writing do not involve the parent directory `"my_files"`. This synchronization requirement applies to all 3 tasks: the buffer cache, extensible files, and subdirectories.

Note: If you added a global file system lock in Project 2, remember to remove it!

2 Deliverables

Your project grade will be made up of 4 components:

- 20% Design Document and Design Review
- 55% Code
- 15% Student Testing
- 10% Final Report and Code Quality

2.1 Design Document (Due 4/22) 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

Write your design document inside the `doc/project3.md` file, which has already been placed in your group's GitHub repository. You must use GitHub Flavored Markdown¹ to format your design document. You can preview your design document on GitHub's web interface by going to the following address: (replace `group0` with your group number)

<https://github.com/Berkeley-CS162/group0/blob/master/doc/project3.md>

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 parts. 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**. This section should be similar in style and format to the design document you submitted for Project 1 and Project 2. We expect you to read through the Pintos source code when preparing your design document, and your design document should refer to the Pintos source when necessary to clarify your implementation.
3. **Synchronization** – This section should list all resources that are shared across threads. For each case, enumerate how the resources are accessed (e.g., from inside of the scheduler, in an interrupt context, etc), and describe the strategy you plan to use to ensure that these resources are shared and modified safely. For each resource, demonstrate that your design ensures correct behavior and avoids deadlock. 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

¹<https://help.github.com/articles/basic-writing-and-formatting-syntax/>

simplify your strategy. Please discuss the time/memory costs of your synchronization approach, and whether your strategy will significantly limit the parallelism of the kernel. When discussing the parallelism 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.

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 Topics for your design document

Make sure to address each of these issues in the **Algorithms** and **Synchronization** sections of your design document. You do not need to answer these questions directly, but your design document should clearly demonstrate that your design will not exhibit any of these problems.

- When one process is actively reading or writing data in a buffer cache block, how are other processes prevented from evicting that block?
- During the eviction of a block from the cache, how are other processes prevented from attempting to access the block?
- If a block is currently being loaded into the cache, how are other processes prevented from also loading it into a different cache entry? How are other processes prevented from accessing the block before it is fully loaded?
- How will your filesystem take a relative path like `../my_files/notes.txt` and locate the corresponding directory? Also, how will you locate absolute paths like `/cs162/solutions.md`?
- Will a user process be allowed to delete a directory if it is the cwd of a running process? The test suite will accept both “yes” and “no”, but in either case, you must make sure that new files cannot be created in deleted directories.
- How will your syscall handlers take a file descriptor, like 3, and locate the corresponding file or directory struct?
- You are already familiar with handling memory exhaustion in C, by checking for a NULL return value from malloc. In this project, you will also need to handle disk space exhaustion. When your filesystem is unable to allocate new disk blocks, you must have a strategy to abort the current operation and rollback to a previous good state.

2.1.3 Design Document Additional Questions

You must also answer this additional question in your design document:

1. For this project, there are 2 optional buffer cache features that you can implement: write-behind and read-ahead. A buffer cache with write-behind will periodically flush dirty blocks to the filesystem block device, so that if a power outage occurs, the system will not lose as much data. Without write-behind, a write-back cache only needs to write data to disk when (1) the data is dirty and gets evicted from the cache, or (2) the system shuts down. A cache with read-ahead will predict which block the system will need next and fetch it in the background. A read-ahead cache can greatly improve the performance of sequential file reads and other easily-predictable file access patterns. Please discuss a possible implementation strategy for write-behind and a strategy for read-ahead. **You must answer this question regardless of whether you actually decide to implement these features.**

2.1.4 Design Review

You will schedule a 20-25 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.

2.1.5 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. You **must** attend a design review in order to get these points. We will try to accommodate any time conflicts, but you should let your TA know as soon as possible.

2.2 Code (Due 5/4)

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.

2.3 Student Testing (Due 5/5)

Pintos already contains a test suite for Project 3, but it does not cover the buffer cache. For this project, you must implement **two** of the following test cases:

- Test your buffer cache's effectiveness by measuring its cache hit rate. First, reset the buffer cache. Open a file and read it sequentially, to determine the cache hit rate for a cold cache. Then, close it, re-open it, and read it sequentially again, to make sure that the cache hit rate improves.
- Test your buffer cache's ability to coalesce writes to the same sector. Each block device keeps a `read_cnt` counter and a `write_cnt` counter. Write a large file byte-by-byte (make the total file size at least 64KB, which is twice the maximum allowed buffer cache size). Then, read it in byte-by-byte. The total number of device writes should be on the order of 128 (because 64KB is 128 blocks).
- Test your buffer cache's ability to write full blocks to disk without reading them first. If you are, for example, writing 100KB (200 blocks) to a file, your buffer cache should perform 200 calls to `block_write`, but 0 calls to `block_read`, since exactly 200 blocks worth of data are being written. (Read operations on inode metadata are still acceptable.) As mentioned earlier, each block device keeps a `read_cnt` counter and a `write_cnt` counter. You can use this to verify that your buffer cache does not introduce unnecessary block reads.

You should focus on writing tests for general buffer-cache features, rather than writing tests for your specific implementation of the buffer cache. You should write your test cases with a minimal set of assumptions about the underlying buffer cache implementation, but you are permitted to make as many basic assumptions about the buffer cache as you need to, since it is very difficult to write buffer cache tests without doing so. Use your good judgement, and create test cases that could potentially be adapted to a different group's project without rewriting the whole thing.

Once you finish writing your test cases, make sure that they get executed when you run "`make check`" in the `pintos/src/filesys/` directory. You will need to prepare a Student Testing Report, which will

help us grade your test cases. Place your Student Testing Report inside `reports/project3.md`, alongside your final report.

Make sure your Student Testing Report contains the following:

- For each of the 2 test cases you write:
 - Provide a description of the feature your test case is supposed to test.
 - Provide an overview of how the mechanics of your test case work, as well as a qualitative description of the expected output.
 - Provide the output of your own Pintos kernel when you run the test case. Please copy the full raw output file from `fileysys/build/tests/fileysys/extended/your-test-1.output` as well as the raw results from `fileysys/build/tests/fileysys/extended/your-test-1.result`.
 - Identify two non-trivial potential kernel bugs, and explain how they would have affected your output of this test case. You should express these in this form: “If your 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. “If your kernel had a syntax error, then this test case would not run.” does not count).
- Tell us about your experience writing tests for Pintos. What can be improved about the Pintos testing system? (There’s a lot of room for improvement.) What did you learn from writing test cases?

We will grade your test cases based on effort. If all of the above components are present in your Student Testing Report and your test cases are satisfactory, you will get full credit on this part of the project.

2.4 Final Report (Due 5/5) and Code Quality

After you complete the code for your project, you will submit a final report. Write your final report inside the `reports/project3.md` file, which has already been placed in your group’s GitHub repository. 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?
- Your Student Testing Report (see the previous section for more details)

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

- Does your code exhibit any major memory safety problems (especially regarding C strings), memory leaks, poor error handling, or race conditions?
- Did you use consistent code style? Your code should blend in with the existing Pintos code. Check your use of indentation, your spacing, and your naming conventions.
- 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?
- Did you re-implement linked list algorithms instead of using the provided list manipulation
- 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 Getting Started

This project is a continuation of the userprog code you implemented in Project 2. You should use your group's Project 2 code as a starting point. The autograder will run some of the userprog tests of Project 2 in addition to the Project 3 file system tests.

3.2 Source Files

In this project, you'll be working with a large number of files, primarily in the `filesys` directory. To help you understand all the code, we've selected some key files and described them below:

directory.c Manages the directory structure. In Pintos, directories are stored as files.

file.c Performs file reads and writes by doing disk sector reads and writes.

filesys.c Top-level interface to the file system.

free-map.c Utilities for modifying the file system's free block map.

fsutil.c Simple utilities for the file system that are accessible from the kernel command line.

inode.c Manages the data structure representing the layout of a file's data on disk.

lib/kernel/bitmap.c A bitmap data structure along with routines for reading and writing the bitmap to disk files.

All the basic functionality of a file system is already in the skeleton code, so that the file system is usable from the start, as you've seen in Project 2. However, the current file system has some severe limitations which you will remove.

3.3 Testing File System Persistence

Until now, each test invoked Pintos just once. However, an important purpose of a file system is to ensure that data remains accessible from one boot to another. Thus, the Project 3 file system tests invoke Pintos twice. During the second invocation, all the files and directories in the Pintos file system are combined into a single file (known as a tarball), which is then copied from the Pintos file system to the host (your development VM) file system.

The grading scripts check the file system's correctness based on the contents of the file copied out in the second run. This means that your project will not pass any of the extended file system tests labeled ***-persistence** until the file system is implemented well enough to support **tar**, the Pintos user program that produces the file that is copied out. The **tar** program is fairly demanding (it requires both

extensible file and subdirectory support), so this will take some work. Until then, you can ignore errors from `make check` regarding the extracted file system.

Incidentally, as you may have surmised, the file format used for copying out the file system contents is the standard Unix `tar` format. You can use the Unix `tar` program to examine them. The tar file for test T is named `T.tar`.

3.4 Requirements

3.4.1 Buffer Cache

Modify the file system to keep a cache of file blocks. When a request is made to read or write a block, check to see if it is in the cache, and if so, use the cached data without going to disk. Otherwise, fetch the block from disk into the cache, evicting an older entry if necessary. **Your cache must be no greater than 64 sectors in size.**

You must implement a cache replacement algorithm that is at least as good as the “clock” algorithm. We encourage you to account for the generally greater value of metadata compared to data. You can experiment to see what combination of accessed, dirty, and other information results in the best performance, as measured by the number of disk accesses. Running pintos from the `filesystems/build` directory will cause a sum total of disk read and write operations to be printed to the console, right before the kernel shuts down.

You can keep a cached copy of the free map permanently in a special place in memory if you would like. It doesn’t count against the 64 sector limit.

The provided inode code uses a “bounce buffer” allocated with `malloc()` to translate the disk’s sector-by-sector interface into the system call interface’s byte-by-byte interface. You should get rid of these bounce buffers. Instead, copy data into and out of sectors in the buffer cache directly.

When data is written to the cache, it does not need to be written to disk immediately. You should keep dirty blocks in the cache and write them to disk when they are evicted and when the system shuts down (modify the `filesystems_done()` function to do this).

If you only flush dirty blocks on eviction or shut down, your file system will be more fragile if a crash occurs. As an optional feature, you can also make your buffer cache periodically flush dirty cache blocks to disk. If you have non-busy waiting `timer_sleep()` from the Project 1 working, this would be an excellent use for it. Otherwise, you may implement a less general facility, but make sure that it does not exhibit busy-waiting.

As an optional feature, you can also implement read-ahead, that is, automatically fetch the next block of a file into the cache when one block of a file is read. Read-ahead is only really useful when done asynchronously. That means, if a process requests disk block 1 from the file, it should block until disk block 1 is read in, but once that read is complete, control should return to the process immediately. The read-ahead request for disk block 2 should be handled asynchronously, in the background.

We recommend integrating the cache into your design early. In the past, many groups have tried to tack the cache onto a design late in the design process. This is very difficult.

3.4.2 Indexed and Extensible Files

The basic file system allocates files as a single extent, making it vulnerable to external fragmentation: it is possible that an n -block file cannot be allocated even though n blocks are free. **Eliminate this problem by modifying the on-disk inode structure.** In practice, this probably means using an index structure with direct, indirect, and doubly indirect blocks. You are welcome to choose a different scheme as long as you explain the rationale for it in your design documentation, and as long as it does not suffer from external fragmentation (as does the extent-based file system we provide).

You can assume that the file system partition will not be larger than 8 MB. You must support files as large as the partition (minus metadata). Each inode is stored in one disk sector, limiting the number of

block pointers that it can contain. Supporting 8 MB files will require you to implement doubly-indirect blocks.

An extent-based file can only grow if it is followed by empty space, but indexed inodes make file growth possible whenever free space is available. **Implement file growth.** In the basic file system, the file size is specified when the file is created. In most modern file systems, a file is initially created with size 0 and is then expanded every time a write is made off the end of the file. Your file system must allow this.

There should be no predetermined limit on the size of a file, except that a file cannot exceed the size of the file system (minus metadata). This also applies to the root directory file, which should now be allowed to expand beyond its initial limit of 16 files.

User programs are allowed to seek beyond the current end-of-file (EOF). The seek itself does not extend the file. Writing at a position past EOF extends the file to the position being written, and any gap between the previous EOF and the start of the `write()` must be filled with zeros. A `read()` starting from a position past EOF returns no bytes.

Writing far beyond EOF can cause many blocks to be entirely zero. Some file systems allocate and write real data blocks for these implicitly zeroed blocks. Other file systems do not allocate these blocks at all until they are explicitly written. The latter file systems are said to support “sparse files.” You may adopt either allocation strategy in your file system.

3.4.3 Subdirectories

Implement support for hierarchical directory trees. In the basic file system, all files live in a single directory. Modify this to allow directory entries to point to files or to other directories.

Make sure that directories can expand beyond their original size just as any other file can.

The basic file system has a 14-character limit on file names. You may retain this limit for individual file name components, or may extend it. **You must allow full path names to be much longer than 14 characters.**

Maintain a separate current directory for each process. At startup, set the file system root as the initial process’s current directory. When one process starts another with the `exec` system call, the child process inherits its parent’s current directory. After that, the two processes’ current directories are independent, so that either changing its own current directory has no effect on the other. (This is why, under Unix, the `cd` command is a shell built-in, not an external program.)

Update the existing system calls so that, anywhere a file name is provided by the caller, an absolute or relative path name may be used. The directory separator character is forward slash (`/`). You must also support special file names `.` and `..`, which have the same meanings as they do in Unix.

Update the `open` system call so that it can also open directories. You **should not** support `read` or `write` on a `fd` that corresponds to a directory. (You will implement `readdir` and `mkdir` for directories instead.) You **should** support `close` on a directory, which just closes the directory.

Update the `remove` system call so that it can delete empty directories (other than the root) in addition to regular files. Directories may only be deleted if they do not contain any files or subdirectories (other than `.` and `..`). You may decide whether to allow deletion of a directory that is open by a process or in use as a process’s current working directory. If it is allowed, then attempts to open files (including `.` and `..`) or create new files in a deleted directory must be disallowed.

Here is some code that will help you split a file system path into its components. It supports all of the features that are required by the tests. It is up to you to decide if and where and how to use it.

```

/* Extracts a file name part from *SRCP into PART, and updates *SRCP so that the
   next call will return the next file name part. Returns 1 if successful, 0 at
   end of string, -1 for a too-long file name part. */
static int
get_next_part (char part[NAME_MAX + 1], const char **srcp) {
    const char *src = *srcp;
    char *dst = part;

    /* Skip leading slashes.  If it's all slashes, we're done. */
    while (*src == '/')
        src++;
    if (*src == '\0')
        return 0;

    /* Copy up to NAME_MAX character from SRC to DST.  Add null terminator. */
    while (*src != '/' && *src != '\0') {
        if (dst < part + NAME_MAX)
            *dst++ = *src;
        else
            return -1;
        src++;
    }
    *dst = '\0';

    /* Advance source pointer. */
    *srcp = src;
    return 1;
}

```

3.4.4 System Calls

Implement the following new system calls:

System Call: `bool chdir (const char *dir)` Changes the current working directory of the process to `dir`, which may be relative or absolute. Returns true if successful, false on failure.

System Call: `bool mkdir (const char *dir)` Creates the directory named `dir`, which may be relative or absolute. Returns true if successful, false on failure. Fails if `dir` already exists or if any directory name in `dir`, besides the last, does not already exist. That is, `mkdir("/a/b/c")` succeeds only if `/a/b` already exists and `/a/b/c` does not.

System Call: `bool readdir (int fd, char *name)` Reads a directory entry from file descriptor `fd`, which must represent a directory. If successful, stores the null-terminated file name in `name`, which must have room for `READDIR_MAX_LEN + 1` bytes, and returns true. If no entries are left in the directory, returns false.

. and .. should not be returned by `readdir`

If the directory changes while it is open, then it is acceptable for some entries not to be read at all or to be read multiple times. Otherwise, each directory entry should be read once, in any order.

`READDIR_MAX_LEN` is defined in `lib/user/syscall.h`. If your file system supports longer file names than the basic file system, you should increase this value from the default of 14.

System Call: `bool isdir (int fd)` Returns true if `fd` represents a directory, false if it represents an ordinary file.

System Call: `int inumber (int fd)` Returns the inode number of the inode associated with `fd`, which may represent an ordinary file or a directory.

An inode number persistently identifies a file or directory. It is unique during the file's existence. In Pintos, the sector number of the inode is suitable for use as an inode number.

We have provided `ls` and `mkdir` user programs, which are straightforward once the above syscalls are implemented. We have also provided `pwd`, which is not so straightforward. The `shell` program implements `cd` internally.

The `pintos extract` and `pintos append` commands should now accept full path names, assuming that the directories used in the paths have already been created. This should not require any significant extra effort on your part.

3.5 Pintos user program tests

You should add your two test cases to the `filesys/extended` test suite, which is included when you run `make check` from the `filesys` directory. All of the `filesys` and `userprog` tests are “user program” tests, which means that they are only allowed to interact with the kernel via system calls. **Since buffer cache information and block device statistics are NOT currently exposed to user programs, you must create new system calls to support your two new buffer cache tests.** You can create new system calls by modifying these files (and their associated header files):

lib/syscall-nr.h Defines the syscall numbers and symbolic constants. This file is used by both user programs and the kernel.

lib/user/syscall.c Syscall functions for user programs

userprog/syscall.c Syscall handler implementations

Some things to keep in mind while writing your test cases:

- User programs have access to a limited subset of the C standard library. You can find the user library in `lib/`.
- User programs cannot directly access variables in the kernel.
- User programs do not have access to `malloc`, since `brk` and `sbrk` are not implemented. User programs also have a limited stack size. If you need a large buffer, make it a static global variable.
- Pintos starts with 4MB of memory and the file system block device is 2MB by default. Don't use data structures or files that exceed these sizes.
- Your test should use `msg()` instead of `printf()` (they have the same function signature).

3.6 How to add tests to Pintos

You can add new test cases to the `filesys/extended` suite by modifying these files:

tests/filesys/extended/Make.tests Entry point for the `filesys/extended` test suite. You need to add the name of your test to the `raw_tests` variable, in order for the test suite to find it.

tests/filesys/extended/my-test-1.c This is the test code for your test (you are free to use whatever name you wish, “my-test-1” is just an example). Your test should define a function called **test_main**, which contains a user-level program. This is the main body of your test case, which should make syscalls and print output. Use the **msg()** function instead of **printf**.

tests/filesys/extended/my-test-1.chk Every test needs a .chk file, which is a Perl script that checks the output of the test program. If you are not familiar with Perl, don’t worry! You can probably get through this part with some educated guessing. Your check script should use the subroutines that are defined in **tests/tests.pm**. At the end, call **pass** to print out the “PASS” message, which tells the Pintos test driver that your test passed.

tests/filesys/extended/my-test-1-persistence.chk Pintos expects a second .chk file for every **filesys/extended** test case. After each test case is run, the kernel is rebooted using the same file system disk image, then Pintos saves the entire file system to a tarball and exports it to the host machine. The ***-persistence.chk** script checks that the tarball of the file system contains the correct structure and contents. **You do not need to do any checking in this file, if your test case does not require it.** However, you should call **pass** in this file anyway, to satisfy the Pintos testing framework.

3.7 Suggested Order of Implementation

To make your job easier, we suggest implementing the parts of this project in the following order. You should think about synchronization throughout all steps.

- Implement the buffer cache and integrate it into the existing file system. At this point all the tests from Project 2 should still pass.
- Extensible files. After this step, your project should pass the file growth tests.
- Subdirectories. Afterward, your project should pass the directory tests.
- Remaining miscellaneous items.
- You can implement extensible files and subdirectories in parallel if you temporarily make the number of entries in new directories fixed.

3.8 FAQ

The following questions have been frequently asked by students in the past.

Can `BLOCK_SECTOR_SIZE` change? No, **`BLOCK_SECTOR_SIZE`** is fixed at 512. For IDE disks, this value is a fixed property of the hardware. Other disks do not necessarily have a 512-byte sector, but for simplicity Pintos only supports those that do.

What is the largest file size that we are supposed to support? The file system partition we create will be 8 MB or smaller. However, individual files will have to be smaller than the partition to accommodate the metadata. You’ll need to consider this when deciding your inode organization.

How should a file name like `a//b` be interpreted? Multiple consecutive slashes are equivalent to a single slash, so this file name is the same as `a/b`.

How about a file name like `../x`? The root directory is its own parent, so it is equivalent to `/x/`.

How should a file name that ends in `/` be treated? Most Unix systems allow a slash at the end of the name for a directory, and reject other names that end in slashes. We will allow this behavior, as well as simply rejecting a name that ends in a slash.

Can we keep a `struct inode.disk` inside `struct inode`? The goal of the 64-block limit is to bound the amount of cached file system data. If you keep a block of disk data—whether file data or metadata—anywhere in kernel memory then you have to count it against the 64-block limit. The same rule applies to anything that’s “similar” to a block of disk data, such as a `struct inode_disk` without the `length` or `sector_cnt` members.

That means you’ll have to change the way the inode implementation accesses its corresponding on-disk inode right now, since it currently just embeds a `struct inode_disk` in `struct inode` and reads the corresponding sector from disk when it’s created. Keeping extra copies of inodes would subvert the 64-block limitation that we place on your cache.

You can store a pointer to inode data in `struct inode`, but if you do so you should carefully make sure that this does not limit your OS to 64 simultaneously open files. You can also store other information to help you find the inode when you need it. Similarly, you may store some metadata along each of your 64 cache entries.

You can keep a cached copy of the free map permanently in memory if you like. It doesn’t have to count against the cache size.

`byte_to_sector()` in `filesystem/inode.c` uses the `struct inode_disk` directly, without first reading that sector from wherever it was in the storage hierarchy. This will no longer work. You will need to change `inode_byte_to_sector()` to obtain the `struct inode_disk` from the cache before using it.